TACLe Summerschool 2016, Yspertal, Austria

Analysing energy consumption by software

Marko van Eekelen & Bernard van Gastel

ICT COST Action IC1202

Monday 9:00 - 10:30: Lecture

Introduction (section 1),
Applications (section 2),
Semantics (section 3),
Basic program transformation (section 4),
Energy-aware program transformation (section 5).

Monday 14:00 - 15:30: Practical

Practical deriving worst case energy bounds.

Tuesday 9:00 - 10:00: Lecture

Over approximation with a Hoare logic (section 6),
Discussion with outlook (section 7).

Ph.D summerschool handout, primarily based on [GKv16], [Ker+14] and [Sch+14]. Draft version, to appear as a technical report.

Marko van Eekelen, Radboud University and Open University of the Netherlands, marko@cs.ru.nl
Bernard van Gastel, Open University of the Netherlands, bvg@ou.nl

CONTENTS

1 INTRODUCTION

CONTEXT *Many aspects of modern life depend on, and are controlled by, software. When you wake up, the alarm clock waking you is controlled by software. During the day, almost every device one uses contains software, like a fridge [Tro15], car, television, elevator, and coffee machine. Because we heavily depend on these devices and since they are everywhere around us, it is important to know the correctness and energy conditions required to function properly. This is strengthened by the fact that software is essential to let our contemporary society function. From tax returns, online theft reports, applications for benefits, to construction work and of course communications, all is handled by software. At this summerschool, methods to analyse energy consumption of software are discussed.*

photo 1 Measuring pollution of a Volkswagen Golf 2.0. Photo: Patrick Pleul/AFP/Getty Images.

As traditionally many savings did occur in the hardware side of a computer, energy consumption is a blind spot when developing software. Each next hardware generation consumed less energy for performing the same amount of work. However, recently this development has lost its pace. At the same time, it becomes more and more clear that software has a huge impact on the behaviour and the properties of devices it runs on. A recent example of software influencing the working of a device is the Volkswagen scandal, where Volkswagen used software to detect if the car was being tested, and if so made the diesel motor exhaust less toxic gases (see photo 1). In [OZH16] it is calculated that 44,000 years of human life are lost in Europe because of the fraud, which lasted at least 6 years. Another example are fridges from Panasonic, which could detect if a test was going on and suppressed energy intensive defrost cycles during this test. Although these are negative examples, it does make clear that the software is in control of the device and its (energy) behaviour.

Although it is evident that software is in control, there is almost no time dedicated in most computer science curricula to energy efficiency of software. This is peculiar since energy is important to modern (software) industry. For years data centers are located at places where the energy is cheap, and since the rise of the smartphone more software engineers recognise that in order to get good user reviews, their software should not obliterate the battery charge of the user's phone. As a result, most students never learn to produce energy efficient code. Software engineers have trouble assessing how much energy will be consumed by their software on a target device, especially when the software is run on a multitude of different systems. With the advent of the internet of things, where software is increasingly embedded in our daily life, it is increasingly important that the **software industry as a whole becomes aware of their energy footprint**, and methods are developed to reduce this footprint.

Furthermore the combination of many individual negative effects can also affect our society at large. Although this effect is less direct, it is not less essential. If devices that are present in large quantities in our society all exhibit the same negative behaviour, like incurring needlessly a too high energy consumption, it can impact public utilities and our economy. Governments increasingly become aware of this societal effect, as indicated by the new laws in the European Union issuing ecodesign requirements for all kind of devices, aiming among others to make them more energy efficient. Examples of product categories having ecodesign requirements are vacuum cleaners, electrical motors, lightning, heaters, cooking appliances [EU-1], televisions [EU-2] and coffee machines [EU-3]. Even requirements leading to relative small improvements in energy efficiency can yield large results at scale, even for devices one typically would not expect to be able to save electricity significantly. Although not necessarily running software, a vacuum cleaner can be such a device. By reducing the energy consumption of vacuum cleaners **alone**, by estimation of the European Union itself [EU-4], **20 TWh** of electricity can be saved in 2020. To put that number into context, the Netherlands used 96.8 TWh of electrical energy in 2013 [EU-5].

During the course of the past years, world wide electricity generation continued to increase, **from 6,131 TWh in 1973 to 23,322 TWh in 2013** [IEA15a], representing a **growth of 280%**. The population growth from 3,909,722,120 to 7,181,715,139 people during the same period was 84% [UN16], therefore we are using more and more energy per capita. Most of the growth in energy generation per capita is attributed to China (97%), and the regional attribution of energy generation shows that Africa, Asia and the Middle East are taking a larger share of the worldwide production. The average energy consumption per capita is distributed unevenly, according to [IEA15a] in 2013 the United States consumed 12,987 kWh/capita of electricity, Germany 7,022 kWh/capita, the Netherlands 6,823 kWh/capita, Spain 5,404 kWh/capita, and Italy 5,124 kWh/capita. But for example only 783 kWh/capita in India and 3778 kWh/capita in China are consumed. As a society we have already difficulty with meeting our energy appetite, this will only increase if people in countries as China

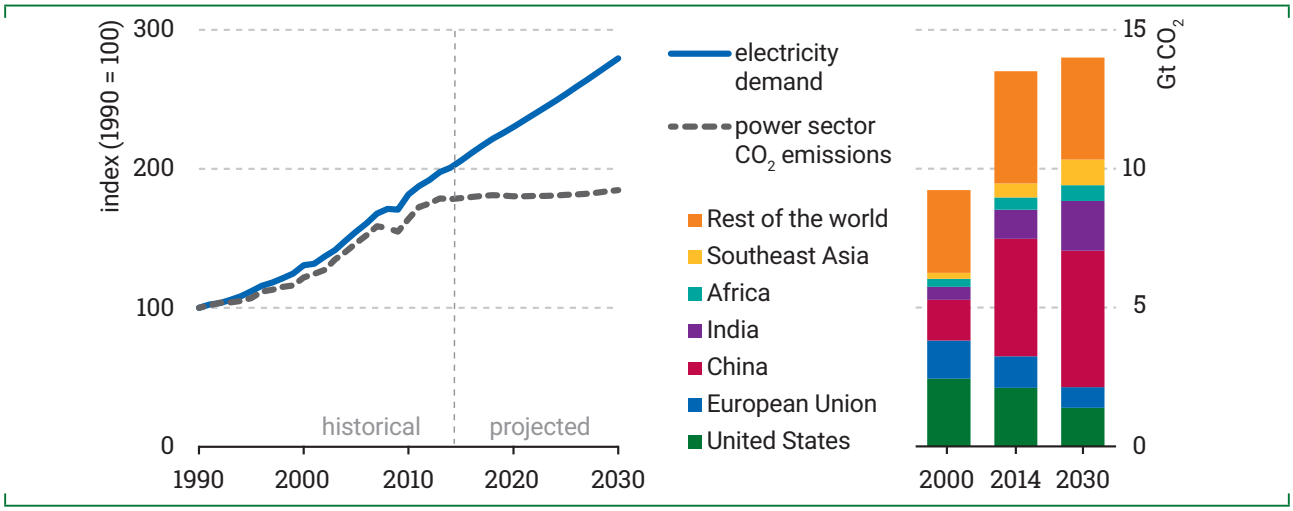

figure 2 Growth in world electricity demand and related CO2 emissions since 1990 (left) and related CO2 emissions by region (right). Source: figure 2 in [IEA15b].

and India are consuming the same amount of energy as in the United States and Europe. According to [IEA15b], the outlook appears the same, with a significant shift in regional CO_2 emissions, as can be found in figure 2.

This context is the basis for the argument of small savings in many devices yielding big societal and environmental impact, devices of which a significant part is running software. We consider a low energy consumption of a device a condition of the software running on the device, as this is required for these devices to be usable in a society at large scale.

1.1 Techniques

A common method the aspects share is **resource analysis**. The method starts with identifying a certain resource, and continue with analysing all the usages of this particular resource. In this lecture the resource studied is **energy**, however other resources can also be studied, such as memory or time. This time resource is needed in order to analyse energy, which we will discuss only briefly, as this will be discussed in other lectures at this summerschool. Resource analysis is a **static analysis method**, meaning the program is analysed without running it. This has a number of advantages as opposed to measuring in a test setup. During the development phase, without actually building the product, the energy usage can be predicted. If the product is actually build, benchmarking energy usage can be expensive, both in terms of energy and cost. This holds especially for large industrial applications, e.g. a whole factory or a large communications infrastructure.

Another common theme is the **transformation of programs**. By transforming the program into another one analysing the program becomes feasible. This transformed program abstracts from all kinds of details, and is equivalent for one particular property. This property can be memory usage or energy consumption.

Another method is, instead of making an transformation, to **over approximate** a certain property. This is a powerful techniques but with its drawbacks. On the one side, if using sound over approximations, allows one derive properties that hold always regardless of the input, i.e. a maximum. These are most of the time symbolic: if the result depend on an input variable, the symbol signifying the input occurs in the result. In order for the results to be correct, stronger assumptions on the input and/or the structure of the program are needed. Another drawback is, as the name implies, imprecision: the derived approximations can be a significant factor worse than the real maximum.

In all these techniques **assumptions** play an important role. Stronger assumptions leads to easier to derive properties and higher quality properties. It is therefore key to clearly identify the properties that are important to a certain case, and make a balanced consideration of which techniques to apply.

1.2 Outline

Lately, research in the relation between energy consumption and software gains traction, with numerous different approaches and views on the subject (see related work later on). We discuss two methods for analysing energy on source code level, one based on a Hoare logic and another based on program transformations.

The first method is an alternative method to analyse programs written in our demonstration language ECA is proposed, using **program transformations**. This language has explicit hardware interaction embedded in the language, enabling easier analysis of the interactions. The program transformation is based on **dependent types**. The approach is hardware parametric, it works for any external hardware device. Several implementations of hardware components can be easily exchanged. This approach has a number of advantages, among others reduced limitations on the modelling of components. This method derives a concrete **precise** result.

The second method deduces an over approximated **symbolic** energy consumption bound from programs written in the ECA language. The **over approximation** is based on a **Hoare logic**, which is applied on the source code level. Due to the nature of over approximations, several extra assumptions on the models are required. The approach is also a hardware parametric The Hoare logic approach suffers both from lack of compositionality and from significant overshoot in the derived bounds.

We focus on software that controls energy consuming hardware, but not necessarily the processor itself as a processor features many implicit behaviour affecting the energy consumption. Examples of control software that can be analysed with the methods in these chapters are control software for thermostats, industrial equipment, environment control systems, and household appliances amongst others.

2 APPLICATIONS

CONTENT

- *defining the scope of the presented approaches.*
- *describing potential use cases.*

The foreseen application area of the proposed analysis is predicting the energy consumption of control systems, in which software controls a number of peripherals. This includes control systems in factories, cars, airplanes, smart-home applications, etc. Examples of hardware components range from a disk drive or sound card, to heaters, engines, motors and urban lighting. The proposed analysis can predict the energy consumption of multiple algorithms and/or different hardware configurations. The choice of algorithm or configuration may depend on the expected workload. This makes the proposed technique useful for both programmers and operators.

The used hardware models can be abstracted, making clear the relative differences between component-methods and component-states. This makes the proposed approach still applicable even when no final hardware component is available for basing the hardware model on, or this model is not yet created. We observe that many decisions are based on relative properties between systems.

In this publication, two manners of deriving energy consumption are used: a type system deriving precise types, and an over approximating Hoare logic. Both work in a syntax-directed manner. Soundness and completeness can be proven by induction on the syntax structure of the program. This proof for the Hoare calculus can be found in [Par+13], and can be repeated for the type system derive precise types, but is more straightforward due to the absence of approximations.

There are certain properties hardware models must satisfy. Foremost, the models are discrete. Implicit state changes by hardware components cannot be expressed. Energy consumption that gradually increases or decreases over time can therefore not be modelled directly. However, discrete approximations may be used.

The quality of the derived energy expressions is directly related to the quality of the used hardware models. Depending on the goal, it is possible to use multiple models for one and the same hardware component. For instance, if the hardware model is constructed as a worst-case model, the type system will produce worst-case information. Similarly one can use average-case models to derive average case information. For the over approximating Hoare logic, worst case hardware models must be used to derive sound bounds.

3 HYBRID MODELLING: LANGUAGE AND SEMANTICS

CONTENT

- *defining language that we will analyse.*
- *define regular semantic of the language.*
- *define hardware models that are used.*
- *define energy enhanced semantics of the language.*

Modern electronic systems typically consist of both hardware and software. As we aim to analyse energy consumption of such **hybrid** systems, we consider hardware and software in a single modelling framework. Software plays a central role, controlling various hardware components. Our analysis works on a simple, special purpose language, called ECA. This language has a special construct for calling functions on hardware components. It is assumed that models exist for all the used hardware components, which model the energy consumption characteristics of these components, as well as the state changes induced by and return values of component function calls.

The ECA language is described in section 3.1. Modelling of hardware components is discussed in section 3.2. Energy-aware semantics for ECA are discussed in section 3.4.

3.1 ECA language

The grammar for the ECA language is defined in listing 3. We presume there is a way to differentiate between identifiers that represents variables ⟨*var*⟩, function names ⟨*function-name*⟩, components ⟨*component*⟩, and constants ⟨*const*⟩.

The only supported type in the ECA language is a signed integer. There are no explicit booleans. The value 0 is handled as **false**, any other value is handled as **true**. The absence of global variables and the by-value passing of variables to functions imply that functions do not have side-effects on the program state. Functions are statically scoped. Recursion is supported in the language (however not in all analysis methods).

The language has an explicit construct for operations on hardware components (e.g. memory, storage or network devices). This allows us to reason about components in a straight-forward manner. Functions on components have a single parameter and always return a value. The notation $C.f$ refers to a function f of a component C.

```
⟨fun-def⟩      ::= 'function' ⟨function-name⟩ '(' ⟨var⟩ ')' 'begin' ⟨expr⟩ 'end'

⟨bin-op⟩       ::= '+' | '-' | '*' | '>' | '>=' | '==' | '!=' | '<=' | '<' | 'and' | 'or'

⟨expr⟩         ::= ⟨const⟩ | ⟨input⟩ | ⟨var⟩
                 | ⟨var⟩ ':=' ⟨expr⟩ | ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
                 | ⟨component⟩ '.' ⟨function-name⟩ '(' ⟨expr⟩ ')'
                 | ⟨function-name⟩ '(' ⟨expr⟩ ')'
                 | ⟨stmt⟩ ';' ⟨expr⟩

⟨stmt⟩         ::= 'skip' | ⟨stmt⟩ ';' ⟨stmt⟩ | ⟨expr⟩
                 | 'if' ⟨expr⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩ 'end'
                 | 'repeat' ⟨expr⟩ 'begin' ⟨stmt⟩ 'end'
                 | 'while' '[' ⟨bound⟩ ']' ⟨expr⟩ 'do' ⟨stmt⟩ 'end'
                 | ⟨fun-def⟩ ⟨stmt⟩
```

listing 3 Grammar for the ECA language.

The language supports a **repeat** construct, which makes the bound an obvious part of the loop and removes the need for evaluating the loop guard with every iteration (as with a **while**). The **repeat** is used for presentation purposes, without loss of generality, since the more commonly used **while** has the same expressive power.

A typical (predictive recursive descent) parser of this language is in the $\mathrm{LL}(2)$ class of parsers, with a small second pass. This second pass is needed, to avoid a possible infinite lookahead that is needed to differentiate between expressions and statements. During the first phase expressions and statements are combined as the same construct. The small post processing step differentiates between the two. This way the language can still be efficiently parsed in a simple manner.

3.2 Hardware component modelling

In order to reason about hybrid systems, we need to model hardware components. A component model consists of a **component state** and a set of **component functions** which operate on the component state. A component model must capture the behaviour of the hardware component with respect to energy consumption. Component models are used to derive dependent types signifying the energy consumption of the modelled hybrid system. The model can be based on measurements or detailed hardware specifications. Alternatively, a generic component model might be used (e.g. for a generic hard disk drive).

A component state is a collection of variables of any type. They can signify e.g. that the component is on, off, or in stand-by. A component function is modelled by two functions: rv_f which produces the return value, and δ_f which updates the (component) state. A component can have multiple component functions. Any state change in components can only occur during a component function call and therefore is made explicit in the source code.

To model energy consumption, each component has a power draw, which depends on the component state. The function ϕ in the component model maps the component state to a power draw. The result of this function is used to calculate **time-dependent energy consumption**.

3.3 Regular semantics

We use a fairly standard semantics for our ECA language, to which we add energy consumption semantics later on. The semantics are listed in figure 4. We explain the used notation below.

The function environment Δ^s (**s** for semantics) contains both the aforementioned component function signatures (for now a pair of a state transition function δ and a return value function $\mathbf{rv}$) and function definitions (pairs of function body e and parameter x). Component states are collected in the component environment Γ. We use the following notation for substitution: $\sigma[x \leftarrow a]$. With $[x \mapsto a]$, we construct a new environment in which x has the value a. We differentiate two kinds of reductions. Expressions reduce from a triple of the expression, program state σ and component state environment Γ, to a triple of a value, a new program state and a new component state environment, with the $\xrightarrow{e}$ operator. As statements do not result in a value, they reduce from a triple of the statement, σ, and Γ to a pair of a new program state and a new component state environment, with the $\xrightarrow{s}$ operator.

There are three rules for the repeat loop. The one labelled **esRepeat** calculates the value of expression e, i.e. the number of iterations for the loop. The **esRepeatLoop** rule then handles each iteration, until the number of remaining iterations is 0. At that point, the evaluation of the loop is ended with the **esRepeatBase** rule. To differentiate it from the normal evaluation rules we use a tuple of the statement to be evaluated and the number of times the statement should be evaluated.

$$\frac{}{\Delta^s \vdash \langle i, \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathcal{I}(i), \sigma, \Gamma\rangle}\ (\text{sInput}) \qquad \frac{}{\Delta^s \vdash \langle c, \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathcal{Z}(c), \sigma, \Gamma\rangle}\ (\text{sConst})$$

$$\frac{}{\Delta^s \vdash \langle x, \sigma, \Gamma\rangle \xrightarrow{e} \langle \sigma(x), \sigma, \Gamma\rangle}\ (\text{sVar})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle e_2, \sigma', \Gamma'\rangle \xrightarrow{e} \langle m, \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma\rangle \xrightarrow{e} \langle n \,\overline{\boxdot}\, m, \sigma'', \Gamma''\rangle}\ (\text{sBinOp})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle x := e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma'\rangle}\ (\text{sAssign})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle}{\Delta^s, C.f = (\delta_f, \mathrm{rv}_f) \vdash \langle C.f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathrm{rv}_f(\Gamma'(C), a), \sigma', \Gamma'[C \leftarrow \delta_f(\Gamma'(C), a)]\rangle}\ (\text{sCmpF})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle e_f, [x \mapsto a], \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta^s, f = (e_f, x) \vdash \langle f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma''\rangle}\ (\text{sCallF})$$

$$\frac{\Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle e_1, \sigma', \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle S_1, e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}\ (\text{sExprConcat})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sExprAsStmt}) \qquad \frac{}{\Delta^s \vdash \langle \mathsf{skip}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma, \Gamma\rangle}\ (\text{sSkip})$$

$$\frac{\Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle S_1; S_2, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sStmtConcat})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-False})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad n \neq 0 \qquad \Delta^s \vdash \langle S_1, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-True})$$

$$\frac{n \leq 0}{\Delta^s \vdash \langle (S_1, n), \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma, \Gamma\rangle}\ (\text{sRepeatBase})$$

$$\frac{n > 0 \qquad \Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n-1), \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle (S_1, n), \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sRepeatLoop})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n), \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{repeat}\ e_1\ \mathsf{begin}\ S_1\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sRepeat})$$

$$\frac{\Delta^s, f = (e_1, x) \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle \mathsf{function}\ f(x)\ \mathsf{begin}\ e_1\ \mathsf{end}\ S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sFuncDef})$$

figure 4 Regular semantics. Note that i stands for an ⟨*input*⟩ term, c for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

3.4 Energy-aware semantics

The regular semantics are extended to include energy consumed by the components while running the program. To this end both the reduction functions $\xrightarrow{e}$ and $\xrightarrow{s}$ are extended so they also return the energy consumed. The full energy-aware semantics are given in figure 5.

Components consume energy in two distinct ways. A component function might directly induce a **time-independent** energy consumption. Apart from that, it can change the state of the component, affecting its **time-dependent** energy consumption. To be able to calculate time-dependent consumption, we need a timing analysis. Depending on your application and desired precision, a Worst-Case Execution Time (WCET) can be integrated, possible in an external fashion. However, we keep it simple for now. Each construct in the language therefore has an associated execution time, for instance t_{if}. Every component function f has a constant time consumption t_f that is part of its function signature, along with δ_f and $\mathbf{rv}_f$, as described in section 3.2. The pair stored in Δ^{es} (**es** for energy semantics) for each component function f is extended to also contain the execution time t_f, and an incidental energy consumption E_f.

As our system allows for a fixed number of components, known before the analysis is performed, we assume a global function Φ is present. This Φ sums the power draw for each known component, by applying the component power draw function ϕ to the component state as contained in Γ.

$$\frac{}{\Delta^{es};\Phi \vdash \langle i,\sigma,\Gamma\rangle \xrightarrow{e} \langle \mathcal{I}(i),\sigma,\Gamma,\Phi(\Gamma)\cdot t_{\text{input}}\rangle}\ (\text{esInput}) \qquad \frac{}{\Delta^{es};\Phi \vdash \langle c,\sigma,\Gamma\rangle \xrightarrow{e} \langle \mathcal{Z}(c),\sigma,\Gamma,\Phi(\Gamma)\cdot t_{\text{const}}\rangle}\ (\text{esConst})$$

$$\frac{}{\Delta^{es};\Phi \vdash \langle x,\sigma,\Gamma\rangle \xrightarrow{e} \langle \sigma(x),\sigma,\Gamma,\Phi(\Gamma)\cdot t_{\text{var}}\rangle}\ (\text{esVar})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle e_2,\sigma',\Gamma'\rangle \xrightarrow{e} \langle m,\sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle e_1 \boxdot e_2,\sigma,\Gamma\rangle \xrightarrow{e} \langle n \,\overline{\boxdot}\, m,\sigma'',\Gamma'',E'+E''+\Phi(\Gamma'')\cdot t_{\boxdot}\rangle}\ (\text{esBinOp})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle x := e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma'[x \leftarrow n],\Gamma',E'+\Phi(\Gamma')\cdot t_{\text{assign}}\rangle}\ (\text{esAssign})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle a,\sigma',\Gamma',E'\rangle}{\Delta^{es},C.f=(\delta_f,\text{rv}_f,t_f,E_f);\Phi \vdash \langle C.f(e_1),\sigma,\Gamma\rangle \xrightarrow{e} \langle \text{rv}_f(\Gamma'(C),a),\sigma',\Gamma'[C \leftarrow \delta_f(\Gamma'(C),a)],E'+\Phi(\Gamma')\cdot t_f+E_f\rangle}\ (\text{esCmpF})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle a,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle e_f,[x \mapsto a],\Gamma'\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E''\rangle}{\Delta^{es},f=(e_f,x) \vdash \langle f(e_1),\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma'',E'+E''\rangle}\ (\text{esCallF})$$

$$\frac{\Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle e_1,\sigma',\Gamma'\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle S_1,e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E'+E''\rangle}\ (\text{esExprConcat})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}\ (\text{esExprAsStmt}) \qquad \frac{}{\Delta^{es};\Phi \vdash \langle \text{skip},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma,\Gamma,0\rangle}\ (\text{esSkip})$$

$$\frac{\Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle S_2,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle S_1;S_2,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esStmtConcat})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle 0,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle S_2,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+\Phi(\Gamma')\cdot t_{\text{if}}+E''\rangle}\ (\text{esIf-False})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad n \neq 0 \qquad \Delta^{es};\Phi \vdash \langle S_1,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+\Phi(\Gamma')\cdot t_{\text{if}}+E''\rangle}\ (\text{esIf-True})$$

$$\frac{n \leq 0}{\Delta^{es};\Phi \vdash \langle (S_1,n),\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma,\Gamma,0\rangle}\ (\text{esRepeatBase})$$

$$\frac{n>0 \qquad \Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle (S_1,n-1),\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle (S_1,n),\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esRepeatLoop})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle (S_1,n),\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{repeat } e_1 \text{ begin } S_1 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esRepeat})$$

$$\frac{\Delta^{es},f=(e_1,x) \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle \text{function } f(x) \text{ begin } e_1 \text{ end } S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}\ (\text{esFuncDef})$$

figure 5 Energy-aware semantics. Note that i stands for an ⟨*input*⟩ term, c for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

4 PRECISE WITH A PROGRAM TRANSFORMATION

CONTENT

- *define basic program transformation, deriving expressions signifying how an input state is modified.*
- *if executed on a concrete input, the derived expressions yield a concrete energy consumption (and output state).*
- *introduces higher order combinator operators to combine derived expressions.*
- *these expressions in itself can be viewed as a symbolic expression.*
- *define energy aware program transformation, deriving expressions if executed result in the energy consumed*

4.1 Basic program transformation

Before we can introduce a dependent type system that can be used to derive energy consumption expressions of an ECA program, we need to define a standard dependent type system to reason about variable values. We separately introduce this for presentation purposes.

Note that, instead of direct expressions over inputs, we derive functions that calculate a value based on values of the input. This allows us to combine these functions using function composition and to reuse them as signatures for methods and parts of the code.

We will start with a series of definitions. A program state environment called $\mathbf{PState}$ is a function from a variable identifier to a value, i.e. $\mathbf{PState} : \mathbf{Var} \mapsto \mathbf{Value}$. Values are of type $\mathbb{Z}$, like the data type in ECA. A component state $\mathbf{CState}$ is collection of states of components that the component function can work on. A **value function** is a function that, given a (program and component) state, will yield a concrete value, i.e. its signature is $\mathbf{PState} \times \mathbf{CState} \rightarrow \mathbf{Value}$. A **state update function** is a function from $\mathbf{PState} \times \mathbf{CState}$ to $\mathbf{PState} \times \mathbf{CState}$. Such a function expresses changes to the state, caused by the execution of statements or expressions.

To make the typing rules more clear, we explain a number of rules in more detail. The full typing rules can be found in figure 6. We will start with the variable access rule:

$$\frac{}{\Delta^v \vdash x : \langle\, \mathbf{lookup}_x \,,\, \mathbf{id} \,\rangle} \text{(btVar)}$$

All rules for evaluation of an expression return a tuple of a function returning the value of the expression when evaluated and a function that modifies the program state and component state. The former one captures the value, i.e. it is a state value function. The latter one captures the effect, i.e. it is a state update

$$\frac{}{\Delta^v \vdash c : \langle\, \mathrm{const}_{\mathcal{N}(c)}\,,\ \mathrm{id}\,\rangle}\ (\text{btConst}) \qquad \frac{}{\Delta^v \vdash x : \langle\, \mathrm{lookup}_x\,,\ \mathrm{id}\,\rangle}\ (\text{btVar})$$

$$\frac{\Delta^v \vdash e_1 : \langle V_1, \Sigma_1\rangle \qquad \Delta^v \vdash e_2 : \langle V_2, \Sigma_2\rangle}{\Delta^v \vdash e_1 \boxdot e_2 : \langle V_1 \,\overline{\boxdot}\, (\Sigma_1 \ggg V_2), \Sigma_1 \ggg \Sigma_2\rangle}\ (\text{btBinOp})$$

$$\frac{\Delta^v \vdash e : \langle V, \Sigma\rangle}{\Delta^v \vdash x := e : \langle V, \Sigma \ggg \mathrm{assign}_x(V)\rangle}\ (\text{btAssign})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ C.f = (x_f, V_f, \Sigma_f) \vdash \\ \quad C.f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg V_f,\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \Sigma_f)\rangle\end{array}}\ (\text{btCmp})$$

$$\frac{\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ f = (x_f, V_f, \Sigma_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(V_f, \mathrm{rec}_V(f)),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(\Sigma_f, \mathrm{rec}_\Sigma(f)))\rangle\end{array}}\ (\text{btCall})$$

$$\frac{\Delta^v, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ f = (x_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_V(f),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_\Sigma(f))\rangle\end{array}}\ (\text{btRec})$$

$$\frac{\Delta^v \vdash S : \Sigma_{st} \qquad \Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\Delta^v \vdash S,e : \langle \Sigma_{st} \ggg V_{ex}, \Sigma_{st} \ggg \Sigma_{ex}\rangle}\ (\text{btExprConcat})$$

$$\frac{\Delta^v \vdash e : \langle V, \Sigma\rangle}{\Delta^v \vdash e : \Sigma}\ (\text{btExprAsStmt}) \qquad \frac{}{\Delta^v \vdash \mathsf{skip} : \mathrm{id}}\ (\text{btSkip})$$

$$\frac{\Delta^v \vdash S_1 : \Sigma_1 \qquad \Delta^v \vdash S_2 : \Sigma_2}{\Delta^v \vdash S_1;S_2 : \Sigma_1 \ggg \Sigma_2}\ (\text{btStmtConcat})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v \vdash S_t : \Sigma_t \qquad \Delta^v \vdash S_f : \Sigma_f}{\Delta^v \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : \mathrm{if}(V_{ex}, \Sigma_{ex} \ggg \Sigma_t, \Sigma_{ex} \ggg \Sigma_f)}\ (\text{btIf})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \mathrm{repeat}^v(V_{ex}, \Sigma_{ex}, \Sigma_{st})}\ (\text{btRepeat})$$

$$\frac{\Delta^v, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \Sigma_{st}}\ (\text{btFuncDef})$$

figure 6 Basic dependent type system. For expressions it yields a tuple of a **value function** and a **state update function**. For statements the type system only derives a **state update function**. Note that c stands for a $\langle const \rangle$ term, x for a $\langle var \rangle$ term, e for a $\langle expr \rangle$ term, S for a $\langle stmt \rangle$ term, $\boxdot$ for a $\langle bin\text{-}op \rangle$, f for a $\langle function\text{-}name \rangle$, and C for a $\langle component \rangle$.

function. To access a variable, we return the **lookup** function (defined below), which is parametrised by the variable that is returned. Variable access does not affect the state, hence for that part the identity function **id** is returned.

The **lookup** function that the **btVar** rule depends on is defined as follows:

$$\begin{aligned}&\mathbf{lookup}_x : \mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}\\&\mathbf{lookup}_x(ps, cs) = ps(x)\end{aligned}$$

Likewise we need to define a function for the **btConst** rule, dealing with constants:

$$\begin{aligned}&\mathbf{const}_x : \mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}\\&\mathbf{const}_x(ps, cs) = x\end{aligned}$$

Before we can continue we need to introduce the $\ggg$ operator. We can compose state update functions with the $\ggg$ operator. Note that the composition is reversed with respect to standard mathematical function composition, in order to maintain the order of program execution. For now, T is $\mathrm{PState} \times \mathrm{CState}$. The $\ggg$ operator can be interpreted as: first apply the effect of the left operand, then execute the right operand on the resulting state. The $\ggg$ operator is defined as:

$$\begin{aligned}&\ggg : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \times (\mathrm{PState} \times \mathrm{CState} \to T)\\&\quad \to (\mathrm{PState} \times \mathrm{CState} \to T)\\&(A \ggg B)(ps, cs) = B(ps', cs') \text{ where } (ps', cs') = A(ps, cs)\end{aligned}$$

Moving on, we can explain the assignment rule below, which assigns a value expressed by expression e to variable x. As an assignment does not modify the value of the expression, the part capturing the value is propagated from the rule deriving the type of x. More interesting is the effect that captures the state change when evaluating the rule. This consists of first applying the state change Σ^v of evaluating the expression e and thereafter replacing the variable x with the result of e (as done by the **assign** function, defined below). This effect is reached with the $\ggg$ operator.

$$\frac{\Delta^v \vdash e : \langle V, \Sigma^v \rangle}{\Delta^v \vdash x := e : \langle V, \Sigma^v \ggg \mathbf{assign}_x(V) \rangle} \text{ (btAssign)}$$

The operator for assigning a new value to a program variable in the type environment:

$$\begin{aligned}&\mathbf{assign}_x : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState})\\&\mathbf{assign}_x(V)(ps, cs) = (ps[x \mapsto V(ps, cs)], cs)\end{aligned}$$

In order to support binary operations we need to define for $\boxdot \in \{+, -, \times, \div, \ldots\}$ a higher order operator $\overline{\boxdot}$. It evaluates the two arguments and combines the results using $\boxdot$. For now, T is $\mathbf{Value}$.

$$\begin{aligned}\overline{\boxdot} : &(\mathrm{PState} \times \mathrm{CState} \to T) \times (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\to (\mathrm{PState} \times \mathrm{CState} \to T) \\ (A \,\overline{\boxdot}\, B)(ps, cs) &= A(ps, cs) \boxdot B(ps, cs)\end{aligned}$$

The binary operation rule can now be introduced. The pattern used in the rule reoccurs multiple times if two expressions (or statements) need to be combined. First the value function representing the first expression, V_1, is evaluated and is combined with the result representing the second expression, V_2, e.g. using the $\overline{\boxdot}$ operator. But this second value function needs to be evaluated in the right context (state): evaluating the first expression can have side-effects and therefore change the program and component state, and influence the outcome of a value function (as the expression can include function calls, assignments, component function calls, etc). To evaluate V_2 in the right context we first must apply the side effects of the first expressions using a state update function Σ_1, expressed as $\Sigma_1 \ggg V_2$. We can express the value function that represents the combination of two expressions as $V_1 \,\overline{\boxdot}\, (\Sigma_1 \ggg V_2)$. The binary operation rule is defined as:

$$\frac{\Delta^v \vdash e_1 : \langle V_1, \Sigma_1 \rangle \qquad \Delta^v \vdash e_2 : \langle V_2, \Sigma_2 \rangle}{\Delta^v \vdash e_1 \boxdot e_2 : \langle V_1 \,\overline{\boxdot}\, (\Sigma_1 \ggg V_2), \Sigma_1 \ggg \Sigma_2 \rangle} \text{ (btBinOp)}$$

Without explaining the rule in detail, we introduce the conditional operator. The **if** operator captures the behaviour of a conditional inside the dependent type. The first argument denotes a function expressing the value of the conditional. For now, T stands for a tuple $\mathrm{PState} \times \mathrm{CState}$.

$$\begin{aligned}\mathbf{if} : &(\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \times (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\times (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\to (\mathrm{PState} \times \mathrm{CState} \to T) \\ \mathbf{if}(c, then, else)(ps, cs) &= \begin{cases} then(ps, cs) & \text{if } c(ps, cs) \neq 0 \\ else(ps, cs) & \text{if } c(ps, cs) = 0 \end{cases}\end{aligned}$$

The **btRepeat** rule can be used to illustrate a more complex rule. The effect of repeat is the composition of evaluating the bound and evaluating the body a number of times. The latter is captured in a **repeat** function that is defined below. The resulting effect can be defined as follows:

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma^v_{ex} \rangle \qquad \Delta^v \vdash S : \Sigma^v_{st}}{\Delta^v \vdash \text{repeat } e \text{ begin } S \text{ end} : \mathbf{repeat}^v(V_{ex}, \Sigma^v_{ex}, \Sigma^v_{st})} \text{ (btRepeat)}$$

The **repeat** operator needed in the **btRepeat** rule captures the behaviour of a loop inside the dependent type. It gets a function that calculates the number of iterations from a type environment, as well as an environment update function for the loop body, and results in an environment update function that represents

the effects of the entire loop. Because the value of the bound must be evaluated in the context before the effect is evaluated, we need an extra function. The actual recursion is in the $\mathbf{repeat}^v$ function, also defined below.

$$\begin{aligned}
&\mathbf{repeat}^v : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\quad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\quad \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\mathbf{repeat}^v(c, start, body)(ps, cs) = \mathbf{repeat}'^v(c(ps, cs), body, ps', cs') \\
&\quad \text{where } (ps', cs') = start(ps, cs)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{repeat}'^v : \mathbb{Z} \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \times \mathrm{PState} \times \mathrm{CState} \\
&\quad \to \mathrm{PState} \times \mathrm{CState} \\
&\mathbf{repeat}'^v(n, body, ps, cs) = \begin{cases} (ps, cs) & \text{if } n \leq 0 \\ \mathbf{repeat}'^v(n-1, body, ps', cs') & \text{if } n > 0 \\ \quad \text{where } (ps', cs') = body(ps, cs) & \end{cases}
\end{aligned}$$

In order to explain the component call rule we need an operator for higher order substitution. This operator creates a new program environment, but retains the component state. It can even update the component state given a Σ function, which is needed because this Σ needs to be evaluated using the original program state. The definition is as follows:

$$\begin{aligned}
&\overline{[x \leftarrow V, \Sigma]} : \mathrm{Var} \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \\
&\quad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\quad \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\overline{[x \leftarrow V, \Sigma]}(ps, cs) = ([x \mapsto V(ps, cs)], cs') \text{ where } (_, cs') = \Sigma(ps, cs)
\end{aligned}$$

We also need an additional operator **split**, because the program state is isolated but the component state is not. The **split** function forks the evaluation into two state update functions and joins the results together. The first argument defines the resulting program state, the second defines the resulting component state.

$$\begin{aligned}
&\mathbf{split} : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\quad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\quad \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\mathbf{split}(A, B)(ps, cs) = (ps', cs') \\
&\quad \text{where } (ps', _) = A(ps, cs) \text{ and } (_, cs') = B(ps, cs)
\end{aligned}$$

The component functions and normal functions are stored in the environment Δ^v. For each function, both component and language ones, a triple is stored, which states the variable name of the argument, a value function that represents the return value, and a state update function that represents the effect on the state of executing the called function. We assume the component function calls are already in the environment. The component function call, listed below, can now be easily expressed:

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, C.f = (x_f, V_f, \Sigma_f) \vdash \\ \quad C.f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg V_f,\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \Sigma_f) \rangle\end{array}} \text{(btCmp)}$$

Like the component call, we can define the function call. However, to support recursion, a special **subst** higher order function is introduced to unfold the function definition once, just before it is executed on a concrete environment. The **subst** is defined as:

$$\begin{aligned}&\mathrm{subst} : (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\qquad \times (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\qquad \to (\mathrm{PState} \times \mathrm{CState} \to T) \\ &\mathrm{subst}(T, R)(ps, cs) = (T[R \leftarrow \mathrm{subst}(T, R)])(ps, cs)\end{aligned}$$

A recursive call is represented by the abstract higher order function **rec**, a sort of placeholder for applying substitution on. There are multiple variants, depending on the resulting type, with the $\mathbf{rec}_V$ one for a resulting value function, and the $\mathbf{rec}_\Sigma$ one for a resulting state update function.

$$\begin{aligned}\mathbf{rec}_V &: \mathrm{PState} \times \mathrm{CState} \to \mathrm{Value} \\ \mathbf{rec}_\Sigma &: \mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}\end{aligned}$$

If there is a function body B computing a **Value** with for example $\mathbf{rec}_V$ in it, the value of the recursive function can be computed by executing $\mathbf{subst}(B, \mathbf{rec}_V)$. As long as the original function terminates on the given input environment, this analysis will terminate on the same input. This is the essential difference from the **btCmp** rule, as can be seen in the definition of **btCall** below:

$$\frac{\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(V_f, \mathbf{rec}_V(f)), \\ \qquad\qquad \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(\Sigma_f, \mathbf{rec}_\Sigma(f))) \rangle\end{array}} \text{(btCall)}$$

For each language function, a definition is placed in Δ^v by means of the **btFuncDef** rule. The body of the function is analysed, and recursive calls to the function are replaced with **rec** placeholders using the **btRec** rule. To support this, the function definition rule inserts a special definition in the function environment Δ^v, on which the **btRec** rule works. This leads to the following definition of **btFuncDef**:

$$\frac{\Delta^v, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash S : \Sigma_{st}}{\Delta^v \vdash \text{function } f(x) \text{ begin } e \text{ end } S : \Sigma_{st}} \text{(btFuncDef)}$$

The placeholders are inserted using the **btRec** rule. This rule analyse the expression used as argument, like the component and function call rules do. The definition is in fact very similar to those definitions:

$$\frac{\Delta^v, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, f = (x_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_V(f),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_\Sigma(f)) \rangle\end{array}} \text{(btRec)}$$

4.2 Energy-aware program transformation

In this section, we present the complete dependent type system, which can be used to determine the energy consumption of ECA programs. The type system presented here should be viewed as an extension of the basic dependent type system in section 4.1. Each time the three dot symbol ... is used, that part of the rule is unchanged with respect to the previous section. The type system adds an element to the results of the previous section, signifying the energy bound. Expressions yield a triple, statements yield a pair. The added element is a function that, when evaluated on a concrete environment and component state, results in a concrete energy consumption or energy cost.

An important energy judgment is the td^{ec} function. To account for time-dependent energy usage, we need this function which calculates the energy usage of all the components over a certain time period. The td^{ec} function is a higher order function that takes the time it accounts for as argument. It results in a function that is just like the other function calculating energy bounds: taking two arguments, a PState and a CState. The definition is given below, which depends on the power draw function ϕ that maps a component state to an energy consumption (as explained in section 3.2):

$$\mathrm{td}^{ec} : \mathrm{Time} \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{EnergyCost})$$
$$\mathrm{td}^{ec}(t)(ps, cs) = t \cdot \sum_{e \in cs} \phi(e)$$

We can use this td^{ec} function to define a rule for variable lookup, as the only energy cost that is induced by variable access is the energy used by all components for the duration of the variable access. This is expressed in the **etVar** rule, in which the dots correspond to figure 5 (the first position is lookup_x, the second id):

$$\frac{}{\Delta^{ec} \vdash x : \langle ..., ..., \mathrm{td}^{ec}(t_{\mathrm{var}}) \rangle} \text{(etVar)}$$

Using the $\overline{+}$ operator (introduced as $\overline{\boxdot}$, with type T equal to $\mathrm{EnergyCost}$), we can define the energy costs of a binary operation. Although the rule looks complex, it uses pattern introduced in section 4.1 for the binary operation. Basically, the energy cost of a binary operation is the cost of evaluating the operands, plus the cost of the binary operation itself. The binary operation itself only has an associated run-time and by this means induces energy consumption of components. This is expressed by the $\mathrm{td}^{ec}(t_{\boxdot})$ function. For the

$$\frac{}{\Delta^{ec} \vdash c : \langle \ldots, \ldots, \mathrm{td}^{ec}(t_{\mathrm{const}}) \rangle}\ (\mathrm{etConst}) \qquad \frac{}{\Delta^{ec} \vdash x : \langle \ldots, \ldots, \mathrm{td}^{ec}(t_{\mathrm{var}}) \rangle}\ (\mathrm{etVar})$$

$$\frac{\Delta^{ec} \vdash e_1 : \langle \ldots, \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle \ldots, \Sigma_2, E_2 \rangle}{\Delta^{ec} \vdash e_1 \boxdot e_2 : \langle \ldots, \ldots, E_1 \mathbin{\overline{+}} (\Sigma_1 \ggg (E_2 \mathbin{\overline{+}} (\Sigma_2 \ggg \mathrm{td}^{ec}(t_{\boxdot})))) \rangle}\ (\mathrm{etBinOp})$$

$$\frac{\Delta^{ec} \vdash e : \langle \ldots, \Sigma, E \rangle}{\Delta^{ec} \vdash x := e : \langle \ldots, \ldots, E \mathbin{\overline{+}} (\Sigma \ggg \mathrm{td}^{ec}(t_{\mathrm{assign}})) \rangle}\ (\mathrm{etAssign})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\begin{array}{l}\Delta^{ec},\ C.f = (x_f, \ldots, \ldots, E_f, t_f) \vdash \\ \quad C.f(e) : \langle \ldots, \ldots, E_{ex} \mathbin{\overline{+}} (\overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg (\mathrm{td}^{ec}(t_f) \mathbin{\overline{+}} E_f)) \rangle\end{array}}\ (\mathrm{etCmp})$$

$$\frac{\Delta^{ec}, f = (x_f, \ldots, \ldots, E_f) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\Delta^{ec}, f = (x_f, \ldots, \ldots, E_f) \vdash f(e) : \langle \ldots, \ldots, E_{ex} \mathbin{\overline{+}} (\overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(E_f, \mathrm{rec}_E(f))) \rangle}\ (\mathrm{etCall})$$

$$\frac{\Delta^{ec}, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\Delta^{ec}, f = (x_f) \vdash f(e) : \langle \ldots, \ldots, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_E(f) \rangle}\ (\mathrm{etRec})$$

$$\frac{\Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle \qquad \Delta^{ec} \vdash e : \langle \ldots, \ldots, E_{ex} \rangle}{\Delta^{ec} \vdash S, e : \langle \ldots, \ldots, E_{st} \mathbin{\overline{+}} (\Sigma_{st} \ggg E_{ex}) \rangle}\ (\mathrm{etExprConcat})$$

$$\frac{\Delta^{ec} \vdash e : \langle \ldots, \Sigma, E \rangle}{\Delta^{ec} \vdash e : \langle \Sigma, E \rangle}\ (\mathrm{etExprAsStmt}) \qquad \frac{}{\Delta^{ec} \vdash \mathsf{skip} : \langle \ldots, \mathrm{zero} \rangle}\ (\mathrm{etSkip})$$

$$\frac{\Delta^{ec} \vdash S_1 : \langle \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash S_2 : \langle \ldots, E_2 \rangle}{\Delta^{ec} \vdash S_1; S_2 : \langle \ldots, E_1 \mathbin{\overline{+}} (\Sigma_1 \ggg E_2) \rangle}\ (\mathrm{etStmtConcat})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S_t : \langle \ldots, E_t \rangle \qquad \Delta^{ec} \vdash S_f : \langle \ldots, E_f \rangle}{\begin{array}{l}\Delta^{ec} \quad \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : \\ \qquad \langle \ldots, E_{ex} \mathbin{\overline{+}} (\Sigma_{ex} \ggg \mathrm{td}^{ec}(t_{\mathrm{if}})) \mathbin{\overline{+}} \mathrm{if}(V_{ex}, \Sigma_{ex} \ggg E_t, \Sigma_{ex} \ggg E_f) \rangle\end{array}}\ (\mathrm{etIf})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \langle \ldots, E_{ex} \mathbin{\overline{+}} \mathrm{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle}\ (\mathrm{etRepeat})$$

$$\frac{\Delta^{ec}, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec}, f = (x, V_{ex}, \Sigma_{ex}, E_{ex}) \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \langle E_{st}, \Sigma_{st} \rangle}\ (\mathrm{etFuncDef})$$

figure 7 Energy-aware dependent type system. It adds an element to the resulting tuples of figure 6 signifying the energy consumption of executing the statement or expression. Note that c stands for a $\langle const \rangle$ term, x for a $\langle var \rangle$ term, e for a $\langle expr \rangle$ term, S for a $\langle stmt \rangle$ term, $\boxdot$ for a $\langle bin\text{-}op \rangle$, f for a $\langle function\text{-}name \rangle$, and C for a $\langle component \rangle$.

binary operation rule we need to apply this pattern twice, in a nested manner, as the time-dependent function must be evaluated in the context after evaluating both arguments. The (three) energy consumptions are added by $\overline{+}$. One can express the binary operation rule as:

$$\frac{\Delta^{ec} \vdash e_1 : \langle ..., \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle ..., \Sigma_2, E_2 \rangle}{\Delta^{ec} \vdash e_1 \boxdot e_2 : \langle ..., ..., E_1 \overline{+} (\Sigma_1 \ggg (E_2 \overline{+} (\Sigma_2 \ggg td^{ec}(t_{\boxdot})))) \rangle} \text{ (etBinOp)}$$

Calculating the energy cost of the repeat loop can be calculated by evaluating an energy cost function for each loop iteration (in the right context). We therefore have to modify the $\mathbf{repeat}^{v}$ function to yield an energy cost, resulting in a new function definition of $\mathbf{repeat}^{ec}$:

$$\begin{aligned}
&\mathbf{repeat}^{ec} : (\text{PState} \times \text{CState} \to \text{Value}) \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \times (\text{PState} \times \text{CState} \to \text{EnergyCost}) \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \to (\text{PState} \times \text{CState} \to \text{EnergyCost}) \\
&\mathbf{repeat}^{ec}(c, start, cost, body)(ps, cs) = \\
&\quad \mathbf{repeat}'^{ec}(c(ps, cs), cost, body, ps', cs') \text{ where } (ps', cs') = \mathbf{start}(ps, cs)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{repeat}'^{ec} : \mathbb{Z} \times (\text{PState} \times \text{CState} \to \text{EnergyCost}) \\
&\quad \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \times \text{PState} \times \text{CState} \to \text{EnergyCost} \\
&\mathbf{repeat}'^{ec}(n, cost, body, ps, cs) = \begin{cases} 0 & \text{if } n \leq 0 \\ \mathbf{repeat}'^{ec}(n-1, cost, body, ps', cs') & \text{if } n > 0 \\ \quad + cost(ps, cs) & \\ \quad\quad \text{where } (ps', cs') = body(ps, cs) & \end{cases}
\end{aligned}$$

Using this $\mathbf{repeat}^{ec}$ function, the definition of the rule for the repeat is analogous to the previous definition.

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \textsf{repeat } e \textsf{ begin } S \textsf{ end} : \langle \ ..., E_{ex} \overline{+} \mathbf{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle} \text{ (etRepeat)}$$

Next is the component function call. The energy cost of a component function call consists of the time taken to execute this function and of explicit energy cost attributed to this call. The environment Δ^{ec} is extended for each component function $C.f$ with two elements: an energy judgment $E_{f'}$ and a run-time $t_{f'}$. Time independent energy usage can be encoded into this $E_{f'}$ function. For functions defined in the language the derived energy judgement is inserted into the environment. There is no need for these functions for an explicit run-time as this is part of the derived energy judgement. Using the patterns described above the component function call is expressed as:

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\begin{array}{ll} \Delta^{ec}, & C.f = (x_f, \ldots, \ldots, E_f, t_f) \vdash \\ & C.f(e) : \langle \ldots, \ldots, E_{ex} \mathbin{\overline{+}} (\overline{[x_f \leftarrow V_{ex}, \Sigma_{ex}]} \ggg (\mathrm{td}^{ec}(t_f) \mathbin{\overline{+}} E_f)) \rangle \end{array}} \text{(etCmpF)}$$

4.3 Example

In this example, we demonstrate our analysis on two programs, comparing their energy usage. Each ECA language construct has an associated execution time bound. These execution times are used in calculating the energy consumption that is time-dependent. Another source of energy consumption in our modelling is time-independent energy usage, which can also be associated with any ECA construct. Adding these two sources together yields the energy consumption.

Consider the example programs in figures 8 and 9. Both programs play #n beeps at #hz Hz (#n and #hz are input variables). The effect of the first statement is starting a component named SoundSystem, which models a sound card (actually, functions have a single parameter; this parameter is omitted here as it is not used). After enabling component SoundSystem, beeps are played by calling the function SoundSystem.playBeepAtHz(). Eventually the device is switched off. The sound system has two states, off $<$ on.

For this example, we will assume a start state in which the component is in the off state and an input $n \in \mathbb{Z}$. Furthermore, the SoundSystem.playBeepAtHz() function has a time-independent energy cost of i, and a call to this function has an execution time of t seconds. The System.sleep() has no associated (time-independent) energy usage, and takes s seconds to complete. The other component function calls and the loop construct are assumed to have zero execution time and zero (time-independent) energy consumption. While switched on, the component has a power draw of u J/s (or W).

We will start with an intuitive explanation of the energy consumption of the program in figure 8, then continue by applying the analysis presented in this paper and comparing these results. Quickly calculating the execution time of the program yields a result of $n \cdot (t + s)$ seconds. As the component is switched on at the start and switched off at the end of the program, the time-dependent energy consumption is $n \cdot (t + s) \cdot u$. The time-independent energy usage is equal to the number of calls to transmit, thus $n \cdot i$, resulting in an energy consumption of $n \cdot (i + (t + s) \cdot u)$ J.

```
1 SoundSystem.on();
2 repeat #n begin
3   SoundSystem.playBeepAtHz(#hz);
4   System.sleep()
5 end;
6 SoundSystem.off()
```

listing 8 Example program.

```
1 repeat #n begin
2   SoundSystem.on();
3   SoundSystem.playBeepAtHz(#hz);
4   SoundSystem.off();
5   System.sleep()
6 end
```

listing 9 Alternative program.

Now, we will show the results from our analysis. Applying the energy-aware dependent typing rules ultimately yields an energy consumption function and a state update function. Both take a typing and a component state environment as input. The first rule to apply is **etStmtConcat**, with the `SoundSystem.on()` call as S_1 and the remainder of the program as S_2. The effect of the component function call, calculated with **etCmpF**, is that the component state is increased (as this signifies a higher time-dependent energy usage). This effect is represented in the component state update function $\delta_{\text{SoundSystem.on}}$. Since we have assumed that the call costs zero time and has no time-independent energy cost, the resulting energy consumption function is the identity function id.

We can now analyse the loop with the **etRepeat** rule. We first derive the type of expression e, which determines the number of iterations of the loop. As the expression is a simple variable access, which we assumed not to have any associated costs, the program state and the component states are not touched. For the result of the expression the type system derives $\mathrm{lookup}_{\#\text{n()}}$ as V_{ex} in the rule, which is (a look-up of) n. This is a function that, when given an input environment, calculates a number in $\mathbb{Z}$ which signifies the number of iterations.

Moving on, we analyse the body of the loop. This means we again apply the **etCmpF** rule to determine the resource consumption of the call to `SoundSystem.playBeepAtHz()`. The call takes time t. Its energy consumption is u if the component is switched `on()`. For ease of presentation, the component states $\{\mathrm{off}, \mathrm{on}\}$ are represented as a variable $e \in \{0,1\}$. The time-independent energy usage of the loop body is i. The energy consumption function E_{st} of the body is $i + e \cdot (t + s) \cdot u$ J.

We can now combine the results of the analysis of the number of iterations and the resource consumption of the loop body (E_{st}) to calculate the consumption of the entire loop. Basically, the resource consumption of the loop E_{st} is multiplied by the number of iterations V_{ex}. This is done in the function $\mathrm{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st})$, in this case $\mathrm{repeat}^{ec}(\mathrm{lookup}_{\#\text{n()}}, \mathrm{id}, E_{st}, \mathrm{id})$. Evaluation after the state update function (corresponding to the `SoundSystem.on()` call), which will update the state of the sound system to `on()`. Evaluating the sequence of both expressions on the start state results in $n \cdot (i + (t + s) \cdot u)$ J, signifying the energy consumption of the code fragment.

Analysing the code fragment listed in figure 9 will result in a type equivalent to $n \cdot (i + t \cdot u)$ J, as the cost of switching the sound system on or off is zero. Given the energy characteristics of the sound system, the second code fragment will have a lower energy consumption. Depending on the realistic characteristics (switching a device on or off normally takes time and energy), a realistic trade-off can be made.

5 OVER APPROXIMATION WITH A HOARE LOGIC

5.1 Changes to energy-aware modelling

Based on the energy-aware models, an upper-bounds for energy consumption of the modelled system is derived. In order for the energy-aware model to be suited for the analysis the model should reflect an upper-bound on the actual consumption. This can be based on detailed documentation or on actual energy measurements.

To provide a sound analysis, we need to assume that components are modelled in such a way that the component states reflect different power-levels and are partially ordered. Larger states should imply greater power draw. We will use finite state models only to enable fix point calculation in our analysis of while loops. The modelling should be such that the following properties hold (in the context of the full soundness proof in [Par+13] these properties are axioms):

components states form a finite lattice with a partial order based on the ordering of polynomials (extended with $\max$) over symbolic variables. Within the lattice each pair of component states has a least upper-bound.

energy-aware component states are partially ordered This ordering extends the ordering on component states in a natural way by adding an energy accumulator and a timestamp. The timestamp stores the time of the latest change to the component state. The earliest timestamp therefore reflects the highest energy usage. Therefore, with respect to timestamps the energy-aware component state ordering should be defined such that smaller timestamps lead to larger energy-aware component states.

power draw functions preserve the ordering i.e. larger states consume more energy than smaller states.

component state update functions δ preserve the ordering For this reason, δ_f cannot depend on the arguments of f. To signify this, we will use $\delta(s)$ in the logic, instead of $\delta(s,a)$. As a result, component models cannot influence each other. Our soundness proof as can be found in [Ker+14; Par+13] requires this assumption.

5.2 Severeness of model restrictions

There are several restrictions to the modelling that may seem far from reality:

component state functions take up a constant amount of time and incidental energy This is needed for the soundness proof. For instance, when a radio component sends a message, the duration of the function call cannot directly depend on the number of bytes in the message. In most cases this can be dealt with by using a different way of modelling. First, one can use an overestimation. Second, such dependencies can be removed by distributing the costs over multiple function calls. For instance, the radio component can have a function to send a fixed number of bytes. If it internally keeps a queue, the additional costs of sending the full queue can be modelled by distributing it over separate queueing operations.

with each component state a constant power draw is associated However, some hardware may accumulate heat over time incurring increasing energy consumption over time. Such a 'heating' problem can be modelled e.g. by changing state to a higher energy level with every call of a component function. This is still an approximation of course. In the future, we want to study models with time driven state change or with time-dependent power draw.

limited dependencies of the component state functions The arguments of a function in the ECA language can not influence the outcome of applying the corresponding state function δ. In other words, the effect of component state functions cannot depend on the arguments of the function in models. Also, component models cannot influence each other. Both restrictions are needed for soundness guarantee of our analysis. This restricts the modelling. Using multiple component state functions instead of dynamic arguments and cross-component calls is a way of modelling that can mitigate these restrictions in certain cases. Relieving these restrictions in general is part of future work.

component model must be finite state machines Modelling systems with finite state machines is not uncommon, e.g using model checking and the right kind of abstraction for the property that is studied. In our models the abstraction should be such that the energy consumption is modelled as close as possible.

These restrictions are due to the desire to over approximate. These limitations do not hamper the precise program transformation presented earlier on. Based on the involved hardware models, one has to make a careful decision on what to use.

EXERCISE 5.1 ***Comparing algorithms with the prototype.*** *We have a prototype of an implementation of the energy analysis tool available at* `http://resourceanalysis.cs.ru.nl/energy/`. *In this exercise we are going to model real world control software and analyse it.*

(a) *Load* `Radio1.eca`. *Study the code, and analyse the program. Modify the program so the radio is switched* `on()` *and* `off()` *in the loop instead of out the loop. Analyse this model, and calculate the break even point of the two algorithms. The break even point is (in this case) the* `N()` *at which one or the other algorithm is more efficient.*

(b) *There is another approach possible given the hardware interface, with buffering and sending items in batches. This is implemented in* `Radio2.eca`*. The variable* `B` *stands for the size of the buffer, and* `K` *is the number of buffer positions left in the current iteration. The annotation* `{K <- B}` *is needed as extra information to make the bound more precise. For a buffer of size 1024, what is the break even point compared to* `Radio1.eca`*?*

(c) *In general terms, what is the break even point if comparing* `Radio1.eca` *and* `Radio2.eca`*? This need to be expressed symbolically (treat* `B` *symbolically). You may have to adjust the cost model in the tab 'radio' and reanalyse the example to quickly discover certain values.*

(d) *How do the constants in the model of the radio influence the break even point in the previous question? Is it possible to decide in a general case (without concrete model variables) which algorithm is best?*

EXERCISE 5.2 ***Modelling and analysing a coffee machine.*** *We are going to model and analyse an espresso machine with a boiler that needs to warm up before making coffee (enabling the boiler incurs a significant incident energy consumption, keeping a high temperature is relatively energy efficient). After a certain amount of minutes it will automatically switch off, to save energy, which is a variable in the algorithm. Make a control algorithm, that for now models user interaction as a parameter indicating the interval in minutes between somebody fetching a coffee. A sleep can be modelled by a component method that takes a certain amount of time, without incidental energy consumption.*

(a) *Implement a model of the coffee machine hardware as used by the algorithm in the prototype. Probably you will have to simplify in order to make it analysable, but keep it a bit realistic.*

(b) *Implement the algorithm and analyse the Worst Case Energy Consumption. How does the standby time impact the energy consumption?*

(c) *Is there any optimisation opportunity in the algorithm?*

(d) *If the users exhibit different behaviour, such as always fetching multiple shots of coffee directly after each other, is this algorithm optimal? Can you specialise the algorithm for such specific user behaviour?*

(e) *Reflect on the process. What can be improved? Is the bound what you expected?*

5.3 A Hoare logic for energy analysis

This section treats the definition of an energy-aware logic with energy analysis rules that can be used to bound the energy consumption of the analysed system. The full set of rules is given in figure 10. These rules are deterministic; at each moment only one rule can be applied. Recursion is currently not supported in this analysis.

Our energy consumption analysis depends on external symbolic analysis of variables and loop analysis. The results of this external analysis are assumed to be accessible in our Hoare logic in two ways.

$$\frac{}{\{\Gamma;t;\rho\}\ n\ \{\Gamma;t;\rho\}}\ \text{(aConst)} \qquad \frac{}{\{\Gamma;t;\rho\}\ x\ \{\Gamma;t;\rho\}}\ \text{(aVar)}$$

$$\frac{\begin{array}{l}\{\Gamma;t;\rho\}\ e_1\ \{\Gamma_1;t_1;\rho_1\} \\ \{\Gamma_1;t_1;\rho_1\}\ e_2\ \{\Gamma_2;t_2;\rho_2\} \qquad \Gamma_2[\mathsf{Imp} \leftarrow \langle \mathrm{s}(\Gamma_2(\mathsf{Imp})), \mathrm{a}(\Gamma_2(\mathsf{Imp})) + E_{\boxdot}, \tau(\Gamma_2(\mathsf{Imp}))\rangle] = \Gamma_3\end{array}}{\{\Gamma;t;\rho\}\ e_1 \boxdot e_2\ \{\Gamma_3; t_2 + t_{\boxdot}; \rho_2\}}\ \text{(aBinOp)}$$

$$\frac{\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\} \qquad \Gamma_1[\mathsf{Imp} \leftarrow \langle \mathrm{s}(\Gamma_1(\mathsf{Imp})), \mathrm{a}(\Gamma_1(\mathsf{Imp})) + E_{\text{assign}}, \tau(\Gamma_1(\mathsf{Imp}))\rangle] = \Gamma_2}{\{\Gamma;t;\rho\}\ x\ \texttt{:=}\ e\ \{\Gamma_2; t_1 + t_{\text{assign}}; \rho_2\}}\ \text{(aAssign)}$$

$$\frac{\Gamma[C \leftarrow \langle \delta_{C.f}(\mathrm{s}(\Gamma(C))), \mathrm{a}(\Gamma(C)) + E_{\mathtt{C.f}} + \mathrm{td}(\Gamma(C), t), t\rangle] = \Gamma_1}{\{\Gamma;t;\rho\}\ C.f\texttt{()}\mathit{args}\texttt{()}\ \{\Gamma_1; t + t_{\mathtt{C.f}}; \rho\}}\ \text{(aCmpF)}$$

$$\frac{\begin{array}{c}\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\} \\ e = a \in \rho \qquad \{\Gamma_1;t_1;\rho_1[x' \leftarrow a]\}\ e_1[x_f \leftarrow x']\ \{\Gamma_2;t_2;\rho_2\} \qquad x' \text{ fresh in } e_1\end{array}}{\Delta, f = (e_1, x_f) \vdash \{\Gamma;t;\rho\}\ f\texttt{(()}e\texttt{)()}\ \{\Gamma_2;t_2;\rho_2\}}\ \text{(aCallF)}$$

$$\frac{}{\{\Gamma;t;\rho\}\ \mathtt{skip}\ \{\Gamma;t;\rho\}}\ \text{(aSkip)} \qquad \frac{\{\Gamma\}\ S\ \{\Gamma_1\} \qquad \{\Gamma_1\}\ e\ \{\Gamma_2\}}{\{\Gamma\}\ S,e\ \{\Gamma_2\}}\ \text{(aExprConcat)}$$

$$\frac{\{\Gamma;t;\rho\}\ S_1\ \{\Gamma_1;t_1;\rho_1\} \qquad \{\Gamma_1;t_1;\rho_1\}\ S_2\ \{\Gamma_2;t_2;\rho_2\}}{\{\Gamma;t;\rho\}\ S_1; S_2\ \{\Gamma_2;t_2;\rho_2\}}\ \text{(aConcat)}$$

$$\frac{\begin{array}{cc}\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\} & \{\Gamma_2;t_1 + t_{\text{if}};\rho_1\}\ S_1\ \{\Gamma_3;t_2;\rho_2\} \\ \Gamma_2 = \Gamma_1[\mathsf{Imp} \leftarrow \langle \mathrm{s}(\Gamma_1(\mathsf{Imp})), \mathrm{a}(\Gamma_1(\mathsf{Imp})) + E_{\text{if}}, \tau(\Gamma_1(\mathsf{Imp}))\rangle] & \{\Gamma_2;t_1 + t_{\text{if}};\rho_1\}\ S_2\ \{\Gamma_4;t_3;\rho_3\}\end{array}}{\{\Gamma;t;\rho\}\ \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \mathtt{end}\ \{\mathrm{lub}(\Gamma_3,\Gamma_4); \max(t_2,t_3); \rho_4\}}\ \text{(aIf)}$$

$$\frac{\begin{array}{cc}\Gamma_1 = \text{process-td}(\Gamma, t) & \Gamma_2[\mathsf{Imp} \leftarrow \langle \mathrm{s}(\Gamma_2(\mathsf{Imp})), \mathrm{a}(\Gamma_2(\mathsf{Imp})) + E_{\text{while}}, \tau(\Gamma_2(\mathsf{Imp}))\rangle] = \Gamma_3 \\ \{\text{wci}(\Gamma_1, e; S); t; \rho\}\ e\ \{\Gamma_2; t_1; \rho_1\} & \{\Gamma_3; t_1 + t_{\text{while}}; \rho_1\}\ S\ \{\Gamma_4; t_2; \rho_2\}\end{array}}{\{\Gamma;t;\rho\}\ \mathtt{while}_{[b]}\ e\ \mathtt{do}\ S\ \mathtt{end}\ \{\text{oe}(\Gamma_1, t, \Gamma_4, t_2, b); \rho_3\}}\ \text{(aWhile)}$$

figure 10 Energy analysis rules.

First, we restrict the scope of our analysis to programs that are bound in terms of execution. We assume that all loops and component functions terminate on any input. Each loop is annotated with a bound: $\texttt{while}_{[b]}$. The bound is a polynomial over the input variables, which expresses an upper-bound on the number of iterations of the loop. We have added the b to the while rule in the energy analysis rules to make this assumption explicit. Derivation of bounds is considered out of scope for our analysis. We assume that an external analysis has produced a sound bound.

Second, the symbolic state environment ρ gives a symbolic state of every variable at each line of code, e.g. $\{x_1 = e_1\}\ x_1 := x_1 + x_2 + x_3\ \{x_1 = e_1 + x_2 + x_3\}$, plus other non-energy related properties invariants that have previously been proven. In figure 10 we included this prerequisite by denoting ρ, ρ_1, … explicitly. As these variables represent external input, thet are not bounded by our logic.

All the judgements in the rules have the following shape: $\{\Gamma; t; \rho\}\ S\ \{\Gamma'; t'; \rho'\}$, where Γ is the set of all energy aware component states, t is the global time and ρ represents the symbolic state environment retrieved from the earlier standard analysis. As (energy-aware) component states are partially ordered, we can take a least upper bound of states $\mathbf{lub}(s_1, s_2)$ and sets of energy-aware component states $\mathbf{lub}(\Gamma_1, \Gamma_2)$.

We will highlight the most relevant aspects of the rules. The **aCmpF** rule uses the $\mathbf{td}(\Gamma(C), t)$ function to estimate the time-dependent energy consumption of component function calls. The **aIf** rule takes the least upper bound of the energy-aware component states and the maximum of the time estimates.

Special attention is warranted for the **aWhile** rule. We study the body of the while loop in isolation. This requires processing the time-dependent energy consumption that occurred before the loop $\mathbf{process\text{-}td}$. An overestimation $\mathbf{oe}$ of the energy consumption of the loop will be calculated by taking the product of the bound on the number of iterations and an overestimation of the energy consumption of a single iteration, i.e. the worst-case iteration $\mathbf{wci}$. The worst-case-iteration is determined by taking the least upper-bound of the set of all states that can occur during the execution of the loop. As there are a finite number of states for each component, this set can be determined via a fix point construction $\mathbf{fix}$. The fix point is calculated by iterating the component iteration function $\mathbf{ci}$.

In order to support the analysis of statements after the loop, also an overestimation of the component states after the loop has to be calculated. For brevity, in figure 10, this is dealt with in the calculation of $\mathbf{oe}$.

Five calculations are needed:

1. Component iteration function $\mathbf{ci}$. The component iteration function $\mathbf{ci}_i(S)$ aggregates the (possibly overestimated) effects of S on C. It performs the analysis on S, then considers only the effects on C. If there are nested loops or conditionals, the effects on the state of C are overestimated in the same manner as in the rest of the analysis. By $\mathbf{ci}_i^n(S)$ we mean the component iteration function applied n times: $\mathbf{ci}_i(S) \circ \mathbf{ci}_i^{n-1}(S)$, with $\mathbf{ci}_i^1(S) = \mathbf{ci}_i(S)$.

2 Fix point function $\mathbf{fix}$. Because component states are finite, there is an iteration after which a component is in a state that it has already been in, earlier in the loop (unless the loop is already finished before this point is reached). Since components are independent, the behaviour of the component will be identical to its behaviour the first time it was in this state. This is a fix point on the set of component states that can be reached in the loop. It can be found using the $\mathbf{fix}_i(S)$ function, which finds the smallest n for which $\exists k.\, \mathbf{ci}_i^{n+1}(S) = \mathbf{ci}_i^k(S)$. The number of possible component states is an upper bound for n.

3 Worst-case iteration function $\mathbf{wci}$. To make a sound overestimation of the energy consumption of a loop, we need to find the iteration that consumes the most. As our analysis is non-decreasing with respect to component states, this is the iteration which starts with the largest component state in the precondition. For this purpose, we introduce the worst-case iteration function $\mathbf{wci}_i(S)$, which computes the least-upper bound of all the states up to the fix point:

$$\mathbf{wci}_i(S) = \mathbf{lub}(\mathbf{ci}_i^0(S), \mathbf{ci}_i^1(S), \ldots, \mathbf{ci}_i^{\mathbf{fix}_i(S)}(S))$$

The global version $\mathbf{wci}(\Gamma, S)$ is defined by iteratively applying the $\mathbf{wci}_i(S)$ function to each component C in Γ.

4 Overestimation function $\mathbf{oe}$. This function overestimates the energy-aware output states of the while loop. It needs to do three things: find the largest non-energy-aware output states, find the minimal timestamps and add the resource consumption of the loop itself. This function gets as input: the start state of the loop Γ_{in}, the start time t, the output state from the analysis of the worst-case iteration Γ_{out}, the end time from the analysis of the worst-case iteration t' and the iteration bound ib. It returns an overestimated energy-aware component state and an overestimated global time.

Because component state update functions δ preserve the ordering, the analysis of the worst-case iteration results in the maximum output state for any iteration. This, however, does not yet address the case where the loop is not entered at all. Therefore, we need to take the least-upper bound of the start state and the result of the analysis of the worst-case iteration.

To overestimate time-dependent energy usage, we must revert component timestamps to the time of entering the loop. So, if a component is switched to a greater state at some point in the loop, the analysis assumes it has been in this state since entering the loop. Note that the least-upper bound of energy-aware component states does exactly this: maximise the non-energy-aware component state and minimise the timestamp. Taking $\Gamma_{base} = \mathbf{lub}(\Gamma_{in}, \Gamma_{out})$ we find both the maximum output states and the minimum timestamps.

Now, we can add the consumption of the loop itself. We perform the following calculation for each component:

$$\mathbf{a}(\Gamma_{base}(C)) = \mathbf{a}(\Gamma_{in}(C)) + b \cdot (\mathbf{a}(\Gamma_{out}(C)) - \mathbf{a}(\Gamma_{in}(C)))$$

We perform something similar for the execution time:

$$t_{ret} = t + b \cdot (t' - t)$$

5 Processing time-dependent energy function $\mathbf{process\text{-}td}$. When analysing an iteration of a loop, we must take care not to include any energy consumption outside of the iteration. This would lead to a large overestimation, since it would be multiplied by the (possibly overestimated) number of iterations. Therefore, before analysing the body, we add the time-dependent energy consumption to the energy accumulator for each component and set all timestamps to the current time. Otherwise, the time-dependent consumption before entering the loop would also be included in the analysis of the iteration. We introduce the function $\mathbf{process\text{-}td}(\Gamma, t)$, which adds $\mathbf{td}(C, t)$ to $\mathbf{a}(C)$ and sets $\tau(C)$ to t, for each component C in Γ.

Applying the rules overestimates the sum of the incidental energy consumption and the time-dependent energy consumption. However, the time-dependent energy consumption is only added to the accumulator at changes of component states. So, as for the energy-aware semantics, the time the components are in their current state should still be accounted for by calculating $\mathbf{a}_{system}(\Gamma_{end}, t_{end})$.

5.4 Example

To illustrate our analysis, we model a wireless sensor node, which has a sensor S and a radio R. We analyse the energy usage of a program that repeatedly measures the sensor for ten seconds, then sends the measurement over the radio, shown in figure 11. The example illustrates both **time-dependent** (sensor) and **incidental** (radio) energy usage. We choose a highly abstract modelling to keep the example brief and simple. A more elaborate example can be found in [Sch+14], where two algorithms are compared using an **implementation** of this Hoare logic. A similar analysis is performed in [Par+13] by hand, comparing two algorithms.

Modelling The sensor component S has two states: s_{on} and s_{off}. It does not have any incidental energy consumption. It has a power draw (thus time-dependent consumption) only when on. For this power draw we introduce the constant $\mathbf{a}_{\text{on}}$. There are two component functions, namely `on()` and `off()`, which switch between the two component states.

The radio component R only has incidental energy consumption. It does not have a state. Its single component function is `send()`, which uses $t_{\text{R.send}}$ time and $E_{\text{R.send}}$ energy.

We also need a timing model. For the sake of presentation, a very simply model is used in which only assignment takes time t_{assign}. The other language constructs are assumed to execute instantly.

Application of the energy-aware semantics from figure 5 on the loop body results in a time consumption t_{body} of $10 + t_{\text{R.send}} + 2 \cdot t_{\text{assign}}$ and an energy consumption $\mathbf{a}_{\text{body}}$ of $10 \cdot \mathbf{a}_{\text{on}} + E_{\text{R.send}}$. Intuitively, the time and energy consumption of the whole loop are $n \cdot t_{\text{body}}$ and $n \cdot \mathbf{a}_{\text{body}}$.

```
while[n] n > 0 do
  S.on();
  ... some code taking 10 seconds ...
  x := S.off();
  R.send(x);
  n := n - 1;
end
```

listing 11 Example program.

Energy consumption analysis The analysis (figure 10) always starts with a symbolic state. Note that only the sensor component S has a state. We introduce the symbol $\mathrm{on}_0^{\mathrm{s}}$ for the symbolic start-state (on or off) of the sensor.

We start the analysis with the while loop. So, we apply the **aWhile** rule:

$$\frac{\begin{array}{c}\Gamma_1 = \text{process-td}(\Gamma_0, t_0) \\ \{\mathrm{wci}(\Gamma_1, n > 0; S_{\mathrm{body}}); t_0; \rho_0\}\ n > 0\ \{\Gamma_2; t_1; \rho_1\} \qquad \{\Gamma_2; t_1; \rho_1\}\ S_{\mathrm{body}}\ \{\Gamma_{it}; t_{it}; \rho_{it}\}\end{array}}{\{\Gamma_0; t_0; \rho_0\}\ \mathtt{while}_{[n]}\ n > 0\ \mathtt{do}\ S_{it}\ \mathtt{end}\ \{\mathrm{oe}(\Gamma_1, t_0, \Gamma_{it}, t_{it}, \rho(n)); t_0 + n \cdot (t_{it} - t_0); \rho_{end}\}}\ \text{(aWhile)}$$

t_{while} was omitted because in the timing model it is equal to 0. We will first solve the process-td and wci functions, then analyse the loop guard and body (i.e. the part above the line), then determine the final results with the oe function.

We first add time-dependent energy consumption and set timestamps to t_0 for all components using the process-td function. If we would not do this, the time-dependent energy consumption **before** the loop would be included in the calculation of the resource consumption of the worst-case iteration. As this would be multiplied by the number of iterations, it would lead to a large overestimation. R does not have a state, so we only need to add the time-dependent consumption of S: $\mathrm{td}(\mathsf{S}, t_0) = \phi_S(\mathrm{on}_0^{\mathrm{s}}) \cdot (t_0 - \tau_0(\mathsf{S}))$, where $\tau_0(\mathsf{S})$ is the symbolic timestamp when starting the analysis.

We must now find the worst-case iteration, using the wci function. For the S component we need the $\mathrm{ci}_s(n > 0; S_{\mathrm{body}})$ function. As the other components do not have a state, $\mathrm{ci}_{imp}(n > 0; S_{\mathrm{body}})$ and $\mathrm{ci}_r(n > 0; S_{\mathrm{body}})$ are simply the identity function. The loop body sets the state of the sensor to s_{off}, independent of the start state. So, $\mathrm{ci}_s(n > 0; S_{\mathrm{body}})$ always results in s_{off}. Now we can find the fix point. In the first iteration, we enter the loop with symbolic state $\mathrm{on}_0^{\mathrm{s}}$. In the second iteration, the loop is entered with state s_{off}. In the third iteration, the loop is again entered with state s_{off}. We have thus found the fix point. Intuitively, this means that, since after any number of iterations the sensor is off, the symbolic start state, in which it is unknown whether the sensor is on or off, yields the worst-case. The worst-case iteration is calculated by:

$$\mathrm{wci}_s(n > 0; S_{\mathrm{body}}) = \mathrm{lub}(\mathrm{ci}_s^0(n > 0; S_{\mathrm{body}}), \mathrm{ci}_s^1(n > 0; S_{\mathrm{body}})) = \mathrm{on}_0^{\mathrm{s}}$$

As there are no costs associated with the evaluation of expressions, the analysis of $n > 0$ using the **aBinOp** rule does not have any effect on the state.

We continue with the loop body, starting with a component function call to `on()`:

$$\frac{\Gamma_3 = \Gamma_2[\mathsf{S} \leftarrow \langle \delta_{S.on}(\mathrm{s}(\Gamma_2(\mathsf{S}))), \mathrm{a}(\Gamma_2(\mathsf{S})) + \mathrm{td}(\Gamma(C), t_1), t_1\rangle]}{\{\Gamma_2; t_1; \rho_1\}\ \mathtt{S.on()}\ \{\Gamma_3; t_1; \rho_2\}}\ \text{(aCmpF)}$$

There is no incidental energy consumption or time consumption associated with the call. We must however add the time-dependent energy consumption to the energy accumulator, by adding $\mathrm{td}(\mathsf{S}, t)$. Since we have just set $\tau(\mathsf{S})$ to t_0 and the evaluation of $n > 0$ costs no time, hence $t_1 = t_0$, $\mathrm{td}(\mathsf{S}, t_1)$ results in 0. The function $\delta_{\mathsf{R.on}}$ produces new component state s_{on}. It also saves the current timestamp to the component state, in order to know when the last state transition happened. For simplicity, we omit the application of the concatenation rule **aConcat** in the following.

After ten seconds of executing other statements (which we assume only cost time, no energy), the sensor is turned off. The call to the function `off()` returns the measurement, which is assigned to x. We must therefore first apply the **aAssign** rule, which adds t_{assign} to the global time. We now apply the **aCmpF** rule to the right-hand side of the assignment, i.e. the call to `S.off()`. This updates the state of the component to s_{off}. It also executes the $\mathrm{td}(\mathsf{S}, t_2)$ function in order to determine the energy cost of the component being on for ten seconds. Because $t_2 = \tau(\mathsf{S}) + 10$ and our model specifies a power draw of a_{on} for s_{on}, this results in $10 \cdot \mathrm{a}_{\mathrm{on}}$. We add this to the energy accumulator of the sensor component.

We apply the **aCmpF** rule again, this time for the `send()` function of the radio. As the transmission costs a fixed amount of energy, all time-dependent constants associated with transmitting are set to zero. The application of **aCmpF** only adds the incidental energy usage specified by $E_{\mathsf{R.send}}$ and the constant time usage $t_{\mathsf{R.send}}$. Finally, we apply **aBinOp**, which adds no costs, and the **aAssign** rule, which again adds t_{assign}.

Analysis of the worst-case iteration results in global time t_{it} and energy-aware component state environment Γ_{it}. Next we apply the overestimation function $\mathrm{oe}(\Gamma_1, t_0, \Gamma_{it}, t_{it}, \rho(n))$. This takes as base the least-upper bound of Γ_1 and Γ_{it}, which in this case is exactly Γ_1 (note that the state of the sensor is overestimated as $\mathrm{on}_0^{\mathrm{s}}$). It then adds the consumption of the worst-case iteration, multiplied by the number of iterations. The worst-case iteration results in a global time of $t_0 + 10 + t_{\mathsf{R.send}} + 2 \cdot t_{\mathrm{assign}}$. So, oe results in a global time t_{end} of $t_0 + n \cdot (10 + t_{\mathsf{R.send}} + 2 \cdot t_{\mathrm{assign}})$. Note that this is equal to the time consumption yielded by the energy-aware semantics.

A similar calculation is made for energy consumption, for each component, before we can calculate a_{system}. In total, the oe function results in an energy usage of $\mathrm{a}_0 + n \cdot (10 \cdot \mathrm{a}_{\mathrm{on}} + E_{\mathsf{R.send}})$. However, we still need to add the time-dependent energy consumption for each component. This is where potential overestimation occurs in this example. Since R does not have a state, we only need to add the time-dependent consumption of S. After the analysis of the loop, the state of the sensor is overestimated as $\mathrm{on}_0^{\mathrm{s}}$. This leads to an overestimation when $\mathrm{on}_0^{\mathrm{s}} = s_{\mathrm{on}} \wedge n > 0$, and otherwise the result is equal to that of the energy-aware semantics. The added consumption is:

$$\mathrm{td}(\mathsf{S}, t_{\mathrm{end}}) = \phi_S(\mathrm{on}_0^{\mathrm{s}}) \cdot (t_{\mathrm{end}} - t_0) = \phi_S(\mathrm{on}_0^{\mathrm{s}}) \cdot n \cdot (10 + t_{\mathsf{R.send}} + 2 \cdot t_{\mathrm{assign}})$$

EXERCISE 5.3 ***Theoretical limitations.***

(a) Why are there two `if()`*-rules in the semantics and only one* `if()`*-rule in the energy analysis rules? (likewise for the* `while()`*).*

(b) In the `while()`*-rule in the Hoare logic, a fix point is needed. Why is the fix point needed?*

(c) In the semantics (not the analysis), is it possible to have models in which:

- *the state change depends on the argument of a component function?*
- *the energy draw function can increase over time*
- *the energy draw function can decrease over time*

(d) Why should component state form a finite lattice in the energy analysis?

6 DISCUSSION

The approaches proposed are a first step in making the software industry more aware of the resources they consume, and therefore more sustainable. The presented methods enable programmers to asses the footprint of their software, in terms of memory and energy.

These approaches show that resource analysis is feasible for energy in a precise or over approximated fashion. These approaches are implemented in prototypes, which demonstrate that these methods can be applied in a practical setting, since these analysis methods neither require a super computer nor huge amounts of resources to perform. Implementation details of the software analysed impact the resource consumption significantly, and this is reflected by the bounds (or types) yielded by applying these methods.

Abstraction and encapsulation are often propagated in computer science as a design philosophy. But they can hinder the insight of programmers in the behaviour of their programs. The proposed analysis methods, and more specifically the accompanying case studies, show that the implementation details have a significant influence on the derived resource consumption bounds. More layers of abstraction and encapsulation means less control, and potentially higher resource consumption.

Returning to the societal impact as mentioned in the introduction, the methods presented in this thesis can help to verify that software does not negatively impact society, not even when large quantities of devices are running the software. When development teams are able to apply these approaches, they may gain additional insight in the behaviour of their software and may asses their energy footprint. Problems may be spotted early in the design and development process, before mass production and distribution of the devices. The ability to analyse these programs in this way is one essential step towards making the software industry more sustainable, however much is yet to be done. The approaches presented in this thesis and contributions of others in this line of research potentially have a high societal relevance.

6.1 Validity

There are several validity constraints to the techniques discussed. Although the derivation of bounds is sound, and the derivation of energy consumption is precise, the quality thereof depends directly on the quality of the component models used. The restrictions on component models are severe, e.g. the energy consumption is assumed to be constant in every state of the component. This is in practice not true for most devices, e.g. the energy consumption can be a function over time. To correctly derive energy

consumption (bounds), component models should be bound on their maximum energy usage by other means. To this end, alternative techniques can potentially be applied, such as model transformations and timed model checking, however this is left for future research.

The construction and validation of component models is hard, as there are several real world practical issues. If basing the component model on specifications from hardware vendors, all kinds of errors in the specification are transferred to the component model. Production errors in the hardware, and eventually the degradation of hardware, can induce erratic energy consumption behaviour that does not conform to the specification/component model. Validating a component model with a test setup is hard, as energy is hard to measure. Small differences in energy consumption are hard to measure correctly, and outside condition like temperature can influence the results greatly. It differs significantly from the other resources discussed in the other chapters, which are measurable with great precision within a computer by the computer itself. Validating if the number of states a component model can be in, is the same number of states a hardware model can be in is a hard problem by itself. With powerful models, this validation process with real hardware can potentially last a life time, as least of the hardware in question.

There is another potential source of derived energy consumption not matching the actual behaviour in a realistic situation. Compiler errors and optimisations can impact the (energy) behaviour of a source program greatly. The compiler has influence on the timing of high level language constructs. The timing constants used for these language constructs should match the upper bound it takes to execute those language constructs. Complicating matters even more is the complex design of modern processors executing the software. Even relatively small embedded microprocessors have features, like a register bypass, which impact the execution timing of statements significantly.

These constraints on modelling hardware components and validity implications should be lifted and investigated in future work to make the technique discusses applicable to general real-world problems. However, depending on the context and the precision needed, the current technique can already be applicable. If the hardware component is relatively simple, a conforming component model can be constructed. Another relevant area for the techniques discussed is to give feedback to a prospective programmer, so during construction of software he can optimise the energy consumption.

6.2 Related work

Much of the research that has been devoted to producing green software is focussed on a very abstract level, defining programming and design patterns for writing energy-efficient code [PBL13; Alb10; Sax10; Ran10; NW01; Siv+02]. In [Bri+13], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [Coh+12] and [Sam+11], a program is divided into 'phases' describing similar behaviour. Based on the behaviour of the software, design level optimisations are proposed to achieve lower energy consumption.

Contrary to the abstract levels of the aforementioned papers, there is also research on producing green software on a very low, hardware-specific level [JML06; Ker+13]. Such analysis methods work on specific architectures ([JML06] on SimpleScalar, [Ker+13] on XMOS ISA-level models), while our approach is hardware-parametric.

Furthermore, there is research on building compilers that optimize code for energy-efficiency. In [Zhu+09], the implementation of several energy optimisation methods, including dynamic voltage scaling and bit-switching minimisation, into the GCC compiler is described and evaluated. Iterative compilation is applied to energy consumption in [GCB05]. In [Meh+97], a technique is proposed in which register labels are encoded alternatively to minimise switching costs. This saves an average of 4.25% of energy, without affecting performance. The human-written assembly code for an MP3 decoder is optimised by hand in [Šim+00], guided by an energy profiling tool based on a proprietary ARM simulator. In [Zha+03], functional units are disabled to reduce (leakage) energy consumption of a VLIW processor. The same is done for an adaptation of a DEC Alpha 21264 in [YLL06]. Reduced bit-width instruction set architectures are exploited to save energy in [SD04]. Energy is saved on memory optimisations in [Joo+02; Ver+06; Jon+09; LC03], while [OYI01; Sap+02] focus on variable-voltage processors.

Sustainability can also be seen as a quality property, as explained in [Lag+15]. A framework called Green is presented in [BC10], allowing programmers to approximate expensive functions and calculate Quality of Service (QoS) statistics. It can thus help leverage a trade-off between performance and energy consumption on the one hand, and QoS on the other.

In [Bri+14], Resource-Utilization Models (RUMs) are presented, which are an abstraction of the resource behaviour of hardware components. The component models in this paper can be viewed as an instance of a RUM. RUMs can be analysed, e.g. with the model checker Uppaal, while we use a dedicated dependent type system. A possible future research direction is to incorporate RUMs into our analysis as component models.

Analyses for consumption of generic resources are built using recurrence relation solving [Alb+11], amortised analysis [HAH11], amortisation and separation logic [Atk10] and a Vienna Development Method style program logic [Asp+07]. The main differences with our work are that we include an explicit hardware model and a context in the form of component states. This context enables the inclusion of power-draw that depends on the state of components..

Several dependently typed programming languages exist, such as Epigram [McB05] and Agda [BDN09]. The Deputy system, which adds dependent typing to the C language, is described in [Con+07]. Dependent types are applied to security in [ML10]. Security types there enforce access control and information flow policies.

6.3 Future work

In future work, we aim to implement this analysis in order to evaluate its suitability for larger systems and validate practical applicability. We intend to experiment with implementations of various derived approximating analyses in order evaluate which techniques / approximations work best in which context.

A limitation is currently that only a system controlled by **one central process** (processor) can be analysed. Modern systems often consists of a network of interacting systems. Therefore, incorporating interacting systems would increase applicability of the approach.

Another future research direction is to expand the supported input language. Currently, recursion is only supported for deriving exact energy consumption given a concrete environment. To add recursion to the over approximating analysis, an approach is to use the function signatures to compose a set of cost relations (a special case of recurrence relations). A recurrence solver can then eliminate the recursion in the resulting function signatures. In order to support data types, a size analysis of data types is needed to enable iteration over data structures, e.g. using techniques similar to [SET13; TSv09].

On the language level the type system is precise, however it does not take into account optimisations and transformations below the language level. This can be achieved by analysing the software on a lower level, for example the intermediate representation of a modern compiler. Another motivation to use such an intermediate representation as the input language is support for (combinations of) many higher level languages. In this way, programs written in and consisting of multiple languages can be analysed. It can also account for optimisations (such as common subexpression elimination, inlining, statically evaluating expressions), which in general reduce the execution time of the program and therefore impact the time-dependent energy usage (calls with side effects like component function calls are generally not optimised).

Scientifically speaking, there is still much work to be done. This includes, but is not limited to:

analysing parallel programs for resource usage Parallel programs have many interesting interactions, for a large part depending on the exact timing. This makes deriving upper bounds on resource analysis difficult, as the precision suffers. This results in a large overshoot, making the derived bound less useful in practice.

integrate timing in the resource bounds Resource usage can be viewed as a function over time. Viewed this way, it enables more insight into the behaviour of the resource usage. Together with information on synchronisation moments, and possible scheduling policies, this opens up the possibility to derive more precise resource bounds by adding bounds of processes in the periods between synchronisation moments.

making the tools applicable in practice The methods are demonstrated with prototypes on limited case studies. These methods are yet to be tested on large scale in a practical setting, by programmers. Ideally, these tools are integrated in an IDE, enabling these tools to be used in a daily workflow of a programmer.

support for programming languages used in practice Not all language constructs used in the latest iterations of programming languages are supported by the analysis methods. There will always be a gap between the latest versions of programming languages and analysis methods, as certain programming constructs are hampering the analysis of programs. Just like `goto` is considered harmful, certain constructs should be avoided if a programmer wants to keep their code analysable by these approaches. Other constructs need to be implemented and/or the analysis methods adjusted to cover these constructs.

symbolic execution for energy Another approach is possible besides the program transformation deriving an exact bound for concrete input and the over approximation resulting in symbolic bounds. By using symbolic execution, a result can be obtained, that is both symbolic and precise. Future research should be focussed on using a different interpretation of the derived types, so these types can become a common foundation to base multiple kinds of analysis on.

validation study A validation study, validating the obtained bounds with measurements from a real setup. This is useful to increase the perceived reliability of the approaches presented in this thesis.

A BIBLIOGRAPHY

[GKv16] Bernard E. van Gastel, Rody W.J. Kersten, and Marko C.J.D. van Eekelen. "**Using dependent types to define energy augmented semantics of programs**". In: *Proceedings of the fourth international workshop on foundational and practical aspects of resource analysis (FOPARA'15)*. LNCS. To appear. Springer, 2016 (cited on page 2).

[Ker+14] Rody W.J. Kersten, Paolo Parisen Toldin, Bernard E. van Gastel, and Marko C.J.D. van Eekelen. "**A Hoare logic for energy consumption analysis**". In: *Proceedings of the third international workshop on foundational and practical aspects of resource analysis (FOPARA'13)*. Volume 8552. LNCS. Springer, 2014, pages 93–109. DOI: 10.1007/978-3-319-12466-7_6 (cited on pages 2, 24).

[Sch+14] Marc Schoolderman, Jascha Neutelings, Rody W.J. Kersten, and Marko C.J.D. van Eekelen. "**ECAlogic: hardware-parametric energy-consumption analysis of algorithms**". In: *Proceedings of the 13th workshop on foundations of aspect-oriented languages*. FOAL'14. Lugano, Switzerland: ACM, 2014, pages 19–22. DOI: 10.1145/2588548.2588553 (cited on pages 2, 30).

[Tro15] Jeroen Trommelen. **Ook 'groene' koelkasten voorzien van sjoemelsoftware**. Volkskrant. Retrieved from http://s.vk.nl/s-a4165935/. October 18th, 2015 (cited on page 2).

[OZH16] Rik Oldenkamp, Rosalie van Zelm, and Mark A.J. Huijbregts. "**Valuing the human health damage caused by the fraud of volkswagen**". In: *Environmental pollution* 212 (2016), pages 121–127. DOI: 10.1016/j.envpol.2016.01.053 (cited on page 2).

[Par+13] Paolo Parisen Toldin, Rody W.J. Kersten, Bernard E. van Gastel, and Marko C.J.D. van Eekelen. **Soundness proof for a Hoare logic for energy consumption analysis**. Technical report ICIS–R13009. Radboud University Nijmegen, Oct. 2013 (cited on pages 6, 24, 30).

[PBL13] Giuseppe Procaccianti, Stefano Bevini, and Patricia Lago. "**Energy efficiency in cloud software architectures**". In: *27th international conference on environmental informatics for environmental protection, sustainable development and risk management, enviroinfo 2013, hamburg, germany, september 2-4, 2013. proceedings*. 2013, pages 291–299 (cited on page 35).

[Alb10] Susanne Albers. "**Energy-efficient algorithms**". In: *Communications of the ACM* 53.5 (2010), pages 86–96 (cited on page 35).

[Sax10] Eric Saxe. "**Power-efficient software**". In: *Communications of the ACM* 53.2 (2010), pages 44–48. DOI: 10.1145/1646353.1646370 (cited on page 35).

[Ran10] Parthasarathy Ranganathan. "**Recipe for efficiency: principles of power-aware computing**". In: *Communications of the ACM* 53.4 (2010), pages 60–67 (cited on page 35).

[NW01] Kshirasagar Naik and David S. L. Wei. "**Software implementation strategies for power-conscious systems**". In: *Mobile networks and applications* 6.3 (2001), pages 291–305. DOI: 10.1023/A:1011487018981 (cited on page 35).

[Siv+02] Anand Sivasubramaniam, Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. "**Designing energy-efficient software**". In: *International parallel and distributed processing symposium*. Los Alamitos, CA, USA: IEEE Computer Society, 2002. DOI: 10.1109/IPDPS.2002.1016580 (cited on page 35).

[Bri+13] Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans, and Mehmet Akşit. "**A design method for modular energy-aware software**". In: *Proceedings of the 28th annual ACM symposium on applied computing*. Coimbra, Portugal: ACM, 2013, pages 1180–1182. DOI: 10.1145/2480362.2480584 (cited on page 35).

[Coh+12] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. "**Energy types**". In: *SIGPLAN notices* 47.10 (Oct. 2012), pages 831–850. DOI: 10.1145/2398857.2384676 (cited on page 35).

[Sam+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "**EnerJ: approximate data types for safe and general low-power computation**". In: *SIGPLAN notices* 46.6 (June 2011), pages 164–174. DOI: 10.1145/1993316.1993518 (cited on page 35).

[JML06] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. "**Estimating the worst-case energy consumption of embedded software**". In: *Proceedings of the 12th IEEE real-time and embedded technology and applications symposium*. IEEE, 2006, pages 81–90 (cited on page 36).

[Ker+13] Steven Kerrison, Umer Liqat, Kyriakos Georgiou, Alejandro Serrano Mena, Neville Grech, Pedro Lopez-Garcia, Kerstin Eder, and Manuel V. Hermenegildo. "**Energy consumption analysis of programs based on XMOS ISA-level models**". In: *LOPSTR'13*. Springer, Sept. 2013 (cited on page 36).

[Zhu+09] Dmitry Zhurikhin, Andrey Belevantsev, Arutyun Avetisyan, Kirill Batuzov, and Semun Lee. "**Evaluating power aware optimizations within GCC compiler**". In: *Grow-2009: international workshop on GCC research opportunities*. 2009 (cited on page 36).

[GCB05] Stefan Valentin Gheorghita, Henk Corporaal, and Twan Basten. "**Iterative compilation for energy reduction**". In: *Journal of embedded computing* 1.4 (2005), pages 509–520 (cited on page 36).

[Meh+97] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh. "**Techniques for low energy software**". In: *ISLPED'97: proceedings of the 1997 international symposium on low power electronics and design*. Monterey, California, United States: ACM, 1997, pages 72–75. DOI: 10.1145/263272.263286 (cited on page 36).

[Šim+00] Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. "**Source code optimization and profiling of energy consumption in embedded systems**". In: *ISSS'00: proceedings of the 13th international symposium on system synthesis*. Madrid: IEEE Computer Society, 2000, pages 193–198. DOI: 10.1145/501790.501831 (cited on page 36).

[Zha+03] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and V. De. "**Compiler support for reducing leakage energy consumption**". In: *DATE'03: proceedings of the conference on design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003 (cited on page 36).

[YLL06] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. "**Compilers for leakage power reduction**". In: *ACM transactions on design automation of electronic systems* 11.1 (2006), pages 147–164. DOI: 10.1145/1124713.1124723 (cited on page 36).

[SD04] Aviral Shrivastava and Nikil Dutt. "**Energy efficient code generation exploiting reduced bit-width instruction set architectures (rISA)**". In: *ASP-DAC'04: proceedings of the 2004 Asia and South Pacific design automation conference*. Yokohama, Japan: IEEE Press, 2004, pages 475–477 (cited on page 36).

[Joo+02] Yongsoo Joo, Yongseok Choi, Hojun Shim, Hyung Gyu Lee, Kwanho Kim, and Naehyuck Chang. "**Energy exploration and reduction of SDRAM memory systems**". In: *DAC'02: proceedings of the 39th annual design automation conference*. New Orleans, Louisiana, USA: ACM, 2002, pages 892–897. DOI: 10.1145/513918.514138 (cited on page 36).

[Ver+06] Manish Verma, Lars Wehmeyer, Robert Pyka, Peter Marwedel, and Luca Benini. "**Compilation and simulation tool chain for memory aware energy optimizations**". In: *SAMOS*. 2006, pages 279–288 (cited on page 36).

[Jon+09] Timothy M. Jones, Michael F. P. O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. "**Energy-efficient register caching with compiler assistance**". In: *ACM transactions on architecture and code optimization* 6.4 (2009), pages 1–23. DOI: 10.1145/1596510.1596511 (cited on page 36).

[LC03] Hyung Gyu Lee and Naehyuck Chang. "**Energy-aware memory allocation in heterogeneous non-volatile memory systems**". In: *ISLPED'03: proceedings of the 2003 international symposium on low power electronics and design*. Seoul, Korea: ACM, 2003, pages 420–423. DOI: 10.1145/871506.871609 (cited on page 36).

[OYI01] Takanori Okuma, Hiroto Yasuura, and Tohru Ishihara. "**Software energy reduction techniques for variable-voltage processors**". In: *IEEE design and test of computers* 18.2 (2001), pages 31–41. DOI: 10.1109/54.914613 (cited on page 36).

[Sap+02] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. "**Energy-conscious compilation based on voltage scaling**". In: *LCTES/SCOPES'02: proceedings of the joint conference on languages, compilers and tools for embedded systems*. Berlin, Germany: ACM, 2002, pages 2–11. DOI: 10.1145/513829.513832 (cited on page 36).

[Lag+15] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. "**Framing sustainability as a property of software quality**". In: *Commun. ACM* 58.10 (2015), pages 70–78. DOI: 10.1145/2714560 (cited on page 36).

[BC10] Woongki Baek and Trishul M. Chilimbi. "**Green: a framework for supporting energy-conscious programming using controlled approximation**". In: *SIGPLAN notices* 45.6 (June 2010), pages 198–209. DOI: 10.1145/1809028.1806620 (cited on page 36).

[Bri+14] Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans, Mehmet Akşit, and Shmuel Katz. "**A tool-supported approach for modular design of energy-aware software**". In: *Proceedings of the 29th annual ACM symposium on applied computing, Gyeongju, Korea*. SAC'14. ACM, Mar. 2014 (cited on page 36).

[Alb+11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. "**Closed-form upper bounds in static cost analysis**". In: *Journal of automated reasoning* 46.2 (Feb. 2011), pages 161–203. DOI: 10.1007/s10817-010-9174-1 (cited on page 36).

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "**Multivariate amortized resource analysis**". In: *POPL'11*. Edited by Thomas Ball and Mooly Sagiv. ACM, 2011, pages 357–370 (cited on page 36).

[Atk10] Robert Atkey. "**Amortised resource analysis with separation logic**". In: *Programming languages and systems*. Edited by Andrew D. Gordon. Volume 6012. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 85–103. DOI: 10.1007/978-3-642-11957-6_6 (cited on page 36).

[Asp+07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. "**A program logic for resources**". In: *Theoretical computer science.* 389.3 (Dec. 2007), pages 411–445. DOI: 10.1016/j.tcs.2007.09.003 (cited on page 36).

[McB05] Conor McBride. "**Epigram: practical programming with dependent types**". In: *Advanced functional programming*. Edited by Varmo Vene and Tarmo Uustalu. Volume 3622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pages 130–170. DOI: 10.1007/11546382_3 (cited on page 36).

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. "**A brief overview of Agda – a functional language with dependent types**". In: *Theorem proving in higher order logics*. Edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Volume 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pages 73–78. DOI: 10.1007/978-3-642-03359-9_6 (cited on page 36).

[Con+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and GeorgeC. Necula. "**Dependent types for low-level programming**". In: *Programming languages and systems*. Edited by Rocco De Nicola. Volume 4421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pages 520–535. DOI: 10.1007/978-3-540-71316-6_35 (cited on page 36).

[ML10] Jamie Morgenstern and Daniel R. Licata. "**Security-typed programming within dependently typed programming**". In: *Proceedings of the 15th ACM SIGPLAN international conference on functional programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pages 169–180. DOI: 10.1145/1863543.1863569 (cited on page 36).

[SET13] Olha Shkaravska, Marko C. J. D. van Eekelen, and Alejandro Tamalet. "**Collected size semantics for strict functional programs over general polymorphic lists**". In: *Foundational and practical aspects of resource analysis - third international workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, revised selected papers*. Edited by Ugo Dal Lago and Ricardo Peña. Volume 8552. Lecture Notes in Computer Science. Springer, 2013, pages 143–159. DOI: 10.1007/978-3-319-12466-7_9 (cited on page 37).

[TSv09] Alejandro Tamalet, Olha Shkaravska, and Marko C.J.D. van Eekelen. "**Size analysis of algebraic data types**". In: *Trends in functional programming*. Edited by Peter Achten, Pieter Koopman, and Marco T. Morazán. Volume 9. Trends in Functional Programming. Intellect, 2009, pages 33–48 (cited on page 37).

[EU-1] European Union. **Energy efficient products**. Published online. Retrieved from http://ec.europa.eu/energy/en/topics/energy-efficiency/energy-efficient-products. May 26th, 2016 (cited on page 3).

[EU-2] European Union. **642/2009 and 1062/2010: televisions**. Published online. Retrieved from http://www.eceee.org/ecodesign/products/televisions. May 26th, 2016 (cited on page 3).

[EU-3] European Union. **Regulation 801/2013: non-tertiary coffee machines (lot 25)**. Published online. Retrieved from http://www.eceee.org/ecodesign/products/Lot25_non_tertiary_coffee_machines. May 26th, 2016 (cited on page 3).

[EU-4] European Union. **Vacuum cleaners**. Published online. Retrieved from https://ec.europa.eu/energy/en/topics/energy-efficiency/energy-efficient-products/vacuum-cleaners. May 26th, 2016 (cited on page 3).

[EU-5] European Union. **Electricity production, consumption and market overview**. Published online. Retrieved from http://ec.europa.eu/eurostat/statistics-explained/index.php/Electricity_production,_consumption_and_market_overview. May 26th, 2016 (cited on page 3).

[IEA15a] International Energy Agency. **Key world energy statistics 2015**. 2015. URL: http://www.iea.org/publications/freepublications/publication/KeyWorld_Statistics_2015.pdf (cited on page 3).

[IEA15b] International Energy Agency. **Energy and climate change**. 2015. URL: http://www.worldenergyoutlook.org/media/news/WEO2015_COP21Briefing.pdf (cited on pages 3, 4).

[UN16] Department of Economic United Nations and Population Division Social Affairs. **World population prospects: the 2015 revision**. 2016. URL: http://esa.un.org/unpd/wpp/Download/Standard/Population/ (cited on page 3).